FP6-004381-MACS

MACS

Multi-sensory Autonomous Cognitive Systems Interacting with Dynamic Environments for Perceiving and Using Affordances

Instrument: Specifically Targeted Research Project (STReP)

Thematic Priority: 2.3.2.4 Cognitive Systems

D4.3.2 A software prototype for affordance support

Due date of deliverable: June 30, 2006
Actual submission date: October 17, 2006

Start date of project: September 1, 2004 Duration: 36 months

Linköpings Universitet (LiU-IDA)

Revision: Version 2

Project co-funded by the European Commission within the Sixth Framework Programme (2002–2006)		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

EU Project MACS

Deliverable D4.3.2

A software prototype for affordance support – The Entity Structure Generation Module (ESGM)

Fredrik Heintz, Patrick Doherty, Björn Wingman, Piotr Rudol, Mariusz Wzorek

Number: **MACS/4/3.2**
WP: *4.3*
Status: *revision, version 2*
Created at: *September 1, 2006*
Revised at: *v2 – October 17, 2006*
Internal rev: *v2 – October 17, 2006*

FhG/AIS	Fraunhofer Institut für Intelligente Analyse- und Informationssysteme, Sankt Augustin, D
JR_DIB	Joanneum Research Graz, A
LiU-IDA	Linköpings Universitet, Linköping, S
METU-KOVAN	Middle East Technical University, Ankara, T
OFAI	Österreichische Studiengesellschaft für Kybernetik, Vienna, A

This research was partly funded by the European Commission's 6th Framework Programme IST Project MACS under contract/grant number FP6-004381. The Commission's support is gratefully acknowledged.

Corresponding author's address:

Prof. Dr. Patrick Doherty
Linköpings Universitet
Dept. of Computer and Info. Science
Linköping 581 83, Sweden

Fraunhofer Institut für Intelligente
Analyse- und Informationssysteme
Schloss Birlinghoven
D-53754 Sankt Augustin
Germany

Tel.: +49 (0) 2241 14-2683
(Co-ordinator)

Contact:
Dr.-Ing. Erich Rome

Joanneum Research
Institute of Digital Image Processing
Computational Perception (CAPE)
Steyrergasse 9
A-8010 Graz
Austria

Tel.: +43 (0) 316 876-1769

Contact:
Dr. Lucas Paletta

Linköpings Universitet
Dept. of Computer and Info. Science
Linköping 581 83
Sweden

Tel.: +46 13 24 26 28

Contact:
Prof. Dr. Patrick Doherty

Middle East Technical University
Dept. of Computer Engineering
Inonu Bulvari
TR-06531 Ankara
Turkey

Tel.: +90 312 210 5539

Contact:
Asst. Prof. Dr. Erol Şahin

Österreichische Studiengesellschaft
für Kybernetik (ÖSGK)
Freyung 6
A-1010 Vienna
Austria

Tel.: +43 1 5336112 0

Contact:
Prof. Dr. Georg Dorffner

Contents

Part I

Background

Chapter 1

Introduction

This document describes the entity structure generation module (ESGM) of the affordance-based robot control architecture developed in the MACS project as described in the Development of an Affordance-based Control Architecture' [2]. The ESGM is implemented using a knowledge processing middleware called DyKnow which has been developed in order to facilitate the creation and management of knowledge in a distributed real-time system.

The document is structured as follows. This first part describes the ESGM, DyKnow, and how the ESGM is implemented by DyKnow. The second part is a step-by-step tutorial on how to use the ESGM to implement a small application for finding and examining objects.

Chapter 2

The Entity Structure Generation Module

2.1 Overview

The purpose of the entity structure generation module (ESGM) is to derive representations of entities in the world in such a way that the result supports an affordance-based robotic agent in achieving its goals. To accomplish this the ESGM must collect and process information from many different data sources such as sensors, both physical and virtual, feature extractors and filters according to the needs of the components that are going to use the result. Since the entities and the information about them are continually evolving the ESGM takes a stream approach and views the information over time as streams or time-series of updates. This means that the ESGM must support not only standard single shot queries to the representation of an entity but also continuous queries which are automatically updated as new information becomes available. Since the needs of the robotic agent will change over time it is important that the information processing can be controlled and adapted accordingly.

According to the specification, the entity structure generation module (ESGM) is intended both as a repository or library of static pre-defined entity structure types and a generation mechanism for both static (pre-defined) and dynamic (defined on-the-fly) entity trajectory structures [2]. The ESGM should also be able to store previously generated entity trajectory structures for later use and comparisons. In essence, the ESGM can collect, structure and process time series data from different data sources in the robotic architecture using user defined sampling rates and other information provided as a policy to the module.

Conceptually the ESGM is part of the perception component in the architecture, but it may be used by any of the components to create, store, and manage knowledge that should be available to other modules or components. The ESGM facilitates packaging the result from sensors and feature extracting filters into entity structures to be used by the learning module and the event and execution monitor module. The output of the filters is assumed to be represented by attribute/value pairs. The ESGM is configurable to handle sensors with different sampling rates and to detect and track only the relevant features needed for a task.

The entities and their aspects, as defined below, are the subject of the representation supported by the entity structure generation module. The definitions are taken from the document "Development of an Affordance-based Control Architecture" [2].

Definition 2.1.1 (Entity) *An entity is any thing, object or individual with cohesive structure. One can distinguish between physical entities or mental entities. These will be called external and internal entities, respectively. An entity consists of aspects.*

Definition 2.1.2 (Aspect) *An aspect is a property or characteristic of an entity, or a relation an entity shares with other entities.*

The rest of this chapter describes how these entities and aspects are represented, how they are computed, and the data types and interfaces used in the implementation. The complete specification of the interfaces is available in Appendix III.

2.2 Data Sources

Since the ESGM does not create any entity structures by itself, it needs to receive primitive information about the entities in the world to create from different data sources. According to the specification the ESGM can receive data from three sources:

- from the Feature Extraction component,
- from raw sensory data without prior feature extraction, and
- from virtual sensors.

There are two ways of making a data source available to the ESGM, either the source pushes the data to the ESGM or the ESGM pulls the data from the source on request. If the source pushes the data itself, then it has full control of what information is available and when it is available. This means that the stream of updates can not be changed by the ESGM, which might actually know the current use of the information. If the ESGM is allowed to pull the data on demand then, on the other hand, it can adapt the data flow to the actual use and can also create several streams with different properties from the same source. The requirement on a data source to support the pull model is that it can provide the current entity structure at any time since it does not know exactly when a request will arrive.

To add a data source using the push mechanism a primitive entity trajectory structure is created which is updated by the data source whenever new entity structures are availble. To add a data source using the pull mechanism a sensor interface has to be registered with the ESGM. This interface is then used to pull the current value from the sensor. The definitions of the methods in the ESGM interface to add and remove sensor interfaces are:

```
void add_sensor_interface(in string sensor,
  in string type_name,
  in SensorInterface iface)
  raises(SensorAlreadyExists);

void remove_sensor_interface(in string sensor)
  raises(NoSuchSensor);
```

2.3 Computational Units

A computational unit encapsulates an algorithm or a computation on entity structures, such as feature extraction and filtering. The implementation of the computational unit is external to the ESGM and is viewed as a black box from its perspective. All it knows is that it can use the computational unit to perform a computation on a set of input entity structures and get entity structures back as a result.

The definitions of the methods in the ESGM interface to add and remove computational units are:

```
void add_comp_unit(in string name,
   in string type_name,
   in ValueSortSeq input_types,
   in CompUnit cu)
  raises(CompUnitAlreadyExists);

void remove_comp_unit(in string name)
  raises(NoSuchCompUnit);
```

2.4 Entity Structures

To represent entities we have to represent their aspects, the values of their aspects, and the entities themselves. The fundamental representation of an entity is the *entity structure*. The aspects are represented by *attributes* and the values by *values*. The definitions of the data types used to represent entities and aspects, taken from the document "Development of an Affordance-based Control Architecture" [2], are the following:

Definition 2.4.1 (Attribute) *An attribute is a data type representing an aspect of an entity. Attributes have types.*

Definition 2.4.2 (Value) *A value is quantity or quality of an attribute. The type of the value set a value belongs to is the value's type.*

An attribute does not represent the value of an aspect since it is dependent on whose entity's aspect it is, but only the aspect itself. For example, the aspect weight may be represented by the instance of the attribute data type $\langle name ='' weight'', type = integer \rangle$. The values of the weight aspect of a particular entity would be represented by an aspect trajectory as defined below.

One particular value set is the set of pointers to entity structures. The values in this set refers to particular entities and are used to create recursive entity structures.

Definition 2.4.3 (Entity Frame) *An entity frame is a data structure representing an entity. It consists of a name and a set of attribute/value pairs.*

The entity frame represents a snapshot of an entity at a particular time-point.

Definition 2.4.4 (Recursive Entity Frame) *A recursive entity frame is an entity frame with at least one attribute value pair where the value is a pointer to another entity frame.*

Definition 2.4.5 (Ground Entity Frame) *A ground entity frame is an entity frame which does not have any attribute value pair where the value is a pointer to another entity frame.*

A ground entity frame may also be called a *non-recursive entity frame*.

Definition 2.4.6 (Entity Structure) *An entity structure is a data structure that is either a recursive entity frame, that is, an entity frame that contains at least one attribute value pair where the value part is a pointer to another entity structure, or a plain entity frame where no values point to other entity structures.*

This means that an entity structure is either a recursive or a non-recursive entity frame.

Based on the definitions above we have to make a number of interpretations in order to implement the ESGM. According to definition 2.4.6 it is a entity frame and according to definition 2.4.3 an entity frame consists of a name and a set of attribute/value pairs. But this is not enough, since an entity structure only represents the entity at a particular time-point we also need to keep track of when the information is valid and when it was created. To do this we introduce two new components, the valid time and the create time. The resulting data type is:

```
struct EntityStructure {
  string name;
  Time valid_time;
  Time create_time;
  AttributeValuePairSeq attributes;
};
```

By separating the time-point when the entity structure is valid and the time-point it was created we can differentiate between two entity structures for the same entity with the same valid time. This is useful if it's allowed to update the knowledge about the entity at a particular time-point. If we remove the create time and it is still allowed to update the entity structure, without changing the valid time, inconsistencies could arise in the system. The reason is that many components might have copies of the entity structure but these copies are made at different time-points, and therefore might have different attribute/value pairs. This is avoided if we have both the create time and the valid time, because then even though the valid time is the same the create time is different. If the granularity of the system clock is fine enough then it will not be possible to update an entity structure twice during a single time-point.

2.4.1 Entity Structure Types

Each entity structure is of a particular type. To represent this type a new data type called the entity structure type is introduced. It consists of a name of the type and set of attributes that all instances of the type must assign a value. The resulting data type is:

```
struct EntityStructureType {
  string name;
  AttributeSeq attributes;
};
```

To allow for some type checking to be made it is required that all entity structure types are registered with the ESGM. It is not possible to create an entity structure unless the ESGM has a definition of its entity structure type. The definition of the methods in the ESGM interface to add and remove entity structure types are:

```
void create_entity_structure_type(in string type_name,
  in EntityStructureType entity_type)
  raises(EntityStructureTypeAlreadyExists);

void destroy_entity_structure_type(in string type_name)
  raises(NoSuchEntityStructureType);
```

Since each entity structure type has a name it is also possible to ask for the type associated with a name. The definition of the method in the ESGM interface to do that is:

```
EntityStructureType
get_entity_structure_type(in string type_name)
  raises(NoSuchEntityStructureType);
```

2.5 Entity Trajectory Structures

To represent the development of an entity over time an entity trajectory structure, as defined below, is used.

Definition 2.5.1 (Entity Trajectory) *An entity trajectory is a data structure representing an entity through time. It consists of a name, a subset of the attributes associated with the entity, a temporal interval and a sequence or continuum of entity (sub)frames ordered relative to a suitable temporal structure.*

Definition 2.5.2 (Aspect Trajectory) *An aspect trajectory is an entity trajectory with a single attribute.*

The trajectory consists of a set of entity structures where each of them represents the entity at a particular time-point. The temporal interval of the entity trajectory is then the interval between the time of the earliest and latest entity structure in the trajectory. All of the entity structures in the sequence must be of the same type, which is also the type of the trajectory. Each trajectory structure has a name which is used to refer to this specific trajectory. It is not allowed to use the same name to refer to more than one trajectory at the same time, but it is allowed to change what trajectory a name refers to. There are currently three types of entity trajectory structures: primitive, sampled, and computed. Each of them is described below.

Since each trajectory has a name it is possible to retrieve a reference to a trajectory based on its name. The interface also allows to retrieve the type of a trajectory, which is the type of each of the structures in the trajectory. The definition of the methods in the ESGM interface to retrieve this information are:

```
EntityTrajectoryStructure
get_entity_trajectory_structure(in string name)
  raises(NoSuchEntityStructure);

EntityStructureType
get_entity_trajectory_structure_type(in string name)
  raises(NoSuchEntityStructure);
```

When the trajectory has served its purpose and is no longer needed, it can be destroyed. When it is deleted the name is also removed and can be reused for new trajectories. It is important to destroy unused trajectories since they use valuable resources as long as they are running. The definition of the method in the ESGM interface to destroy a trajectory is:

```
void
destroy_entity_trajectory_structure(in string name)
  raises(NoSuchEntityStructure);
```

2.5.1 Primitive Entity Trajectory Structures

A primitive entity trajectory structure represents an entity where the information about it is provided by a data source outside the ESGM. The external data source must push the entity structures to the trajectory when they are available. This also means that the ESGM has no control of the content of the trajectory. To create a primitive entity trajectory, which is empty to begin with, the ESGM needs to know the name of it and the type of its entity structures. After creation the trajectory is ready to accept new entity structures.

The definition of the method in the ESGM interface to create a primitive entity trajectory structure is:

```
EntityTrajectoryStructure
create_primitive_entity_trajectory_structure(in string name,
     in string type_name)
  raises(EntityStructureAlreadyExists,
 NoSuchEntityStructureType);
```

2.5.2 Sampled Entity Trajectory Structures

A sampled entity trajectory structure represents an entity where the information is provided by a sensor which can be sampled at regular periods. The sensor must be able to read the current entity structure at any time-point since the sensor does not control when it will be called. This is instead controlled by the ESGM. The benefit is that many trajectories, with different sample periods, can be created from the same sensor. To create a sampled entity trajectory structure the ESGM needs to know the name of it, the name of the sensor to sample, the start and end time of the sampling, the sample period, the allowed delay, and the number of entity structures to cache. At the start time of the newly created trajectory the ESGM will start sampling the sensor with the specified sample period until the end time. At every time-point the trajectory will contain at most the cache size number of entity structures. If a new entity structure is added when the cache limit has been reached the oldest entity structure is removed.

The definition of the method in the ESGM interface to create a sampled entity trajectory structure is:

```
EntityTrajectoryStructure
create_entity_trajectory_structure(in string name,
   in string sensor,
   in Idl::Time from,
   in Idl::Time to,
   in Idl::Time sample_period,
```

```
   in Idl::Time delay,
   in long cache_size)
  raises(EntityStructureAlreadyExists,
 NoSuchSensor,
 IllegalPolicy);
```

2.5.3 Computed Entity Trajectory Structures

A computed entity trajectory structure represents an entity which is computed by a computational unit based on one or more entity trajectories. Each time one of the input trajectories is updated, i.e. a new entity structure is added, the computational unit will be called with the new information. Based on its current inputs the computational unit can create a new entity structure which would then be added to the output trajectory of the computational unit. To create a computed entity trajectory structure the ESGM needs to know the name of it, the name of the computational unit to call, and the names of the input trajectories. When created the ESGM will start subscribing to updates to the input trajectories and call the computational unit each time one of them is updated.

The definition of the method in the ESGM interface to create sampled entity trajectory structure is:

```
EntityTrajectoryStructure
create_computed_entity_trajectory_structure(in string name,
    in string cu,
    in StringSeq inputs)
  raises(EntityStructureAlreadyExists,
 NoSuchCompUnit);
```

Chapter 3

DyKnow

3.1 Introduction

The main purpose of DyKnow is to provide generic and well-structured software support for the processes involved in generating state, object, and event abstractions about the environments of complex systems. Generation of state, object, and event representations is done at many levels of abstraction beginning with low level quantitative sensor data and often resulting in qualitative data structures which are grounded in the world and can be interpreted as knowledge by the system. To produce these structures the system has to support operations on data and event streams at many different levels of abstraction. For the result to be useful, the processing must be done in a timely manner so that the robotic agent can react in time to changes in the environment. The resulting structures are used by various functionalities in a deliberative/reactive architecture for control, situation awareness and assessment, monitoring, and planning to achieve mission goals. DyKnow provides a declarative language for specifying these structures needed by the different components of the agent. Based on this specification it creates representations of the external world and the internal state of an agent based on observations and a priori knowledge, such as facts stored in databases, with the desired properties and quality of service guarantees. DyKnow also allows easy integration of existing sensors, databases, reasoning engines and other knowledge producing services.

The motivation for the work is to provide an agent with the necessary knowledge to make decisions and act. This is why we call it knowledge processing middleware instead of e.g. information processing middleware. Since the purpose is acting the processing time is an important part of the knowledge process. It is not sufficient to produce the best possible estimate of the position of the agent if it takes an hour to do it. Neither is it sufficient to produce a fast and dirty estimation of the position if it is not accurate enough to be useful for the agent. The two conditions are necessary, but not sufficient on their own. An important problem is to provide tools to help the agent designer describe and make the trade-offs between efficiency and accuracy.

3.2 Overview

Conceptually, a knowledge processing middleware processes, like DyKnow, streams generated by different components in a distributed system. These streams may be viewed as representations of time-series data and may start as continuous streams from sensors or sequences of queries to databases. These time-series can be interpreted as knowledge by the agent based on the justi-

fication provided by the description of the knowledge producing process. Eventually, they will contribute to definitions of more refined, composite, knowledge structures. Knowledge producing processes combine such streams by computing, synchronizing, filtering and approximating as we move to higher levels of abstraction. In this sense, a knowledge processing middleware supports conventional data fusion processes, but also less conventional qualitative processing techniques common in the area of artificial intelligence. A knowledge producing process has different quality of service properties such as maximum delay, trade-off between quality and delay, how to calculate missing values and so on, which together define the semantics of the knowledge derived by the process. The same streams of data may be processed differently by different parts of the architecture by tailoring the knowledge processes relative to the needs and constraints associated with the tasks at hand.

To separate the issues DyKnow has two levels, the *representational level* and the *symbolic level*. The four main concepts are descriptor, policy, representational structure and represented entity. On the symbolic level the concepts are purely symbolic with a declarative language for defining them, while on the representational level the concepts are actual data structures implemented in a computer. At the same time, the symbolic level is the specification of the representational level, since it defines the expected properties of the data structures. Figure 3.1 shows the top-level concepts and how they are related to each other. On the representational level, a *representational structure* represents a *represented entity* based on observations and a priori knowledge. A representational structure is the result of a knowledge producing process and encapsulates the knowledge about an entity in a structure which can be interacted with. Each representational structure is described by a *policy*, which declares the properties of that representation. A policy is a description, on the symbolic level, of the representational structure. A *descriptor* refers to a policy, also on the symbolic level. A descriptor is also a proxy to the representation since it provides an interface to the underlying representational structure. The meaning of a descriptor is defined by the policy which it refers to. A descriptor is used to interact with the representational structures they denote either to declare more abstract policies and thereby representational structures or by components external to DyKnow. To define a recursive representational structure a policy can contain descriptors which are references to other representational structures.

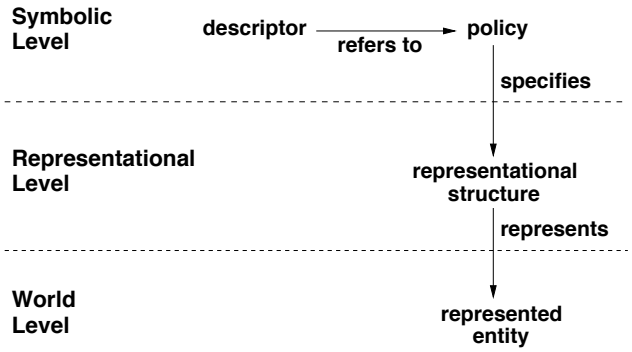

Figure 3.1: An overview of the top-level DyKnow concepts and how they are related.

Each descriptor, policy, representational structure and entity has a type. There are currently four types of entities: objects, events, functions, and features. Each of these have corresponding descriptors, policies and representational structures. In fact features have two different representational structures associated with them. Each of these structures have their own policies and

interfaces. The reason for having different representational structures for the same represented entity is that there are many different ways of structuring the same knowledge about a feature. In our case we have both a structure which views the feature as a total function from time to value and a structure which views the feature as a stream of values. These representational structures have very different properties and interfaces which makes them useful in different applications. In fact, in some cases it is relevant to use both structures for the same feature.

3.2.1 Ontology

The purpose of DyKnow is to create, continually update, and manage representations of entities with well-defined properties. A representation has two parts, the structure and the content. The representational structure is how the representational content is represented. The representational content is what is being represented. For example, the velocity of a robot is a property whose value could either be produced by a sensor or a computational process outside DyKnow, for example by using a derivation function and feed it with the position of the robot. These are two different ways to create the content of the representation of the velocity of the robot, many others could be envisioned as well. In either case we have to think about the structure of the representation, i.e. what are the properties of the representational structure itself. For example, is it possible to query the representation at any time-point or is it only possible to ask for the latest value? Another issue is how much may the value at a time-point be delayed? Both the available interfaces to and the properties of the representational structure are important aspects with respect to a concrete knowledge processing application.

An ontology defines what entities exists and their properties [4]. There are three classes of entities in the ontology. First, we have the entities that DyKnow is designed to model called the represented entities. Second, we have the representational structures which are the data structures that represents these entities. Finally, we have the concepts and terms used in DyKnow to describe representational structures.

3.2.1.1 Represented Entities

A represented entity is anything that we want to represent. An entity may be a physical or an abstract phenomenon which somehow, directly or indirectly, can be observed or reasoned about by the agent. DyKnow currently considers three different kinds of entities: objects, features, and events.

For modelling purposes, we view the environment of the agent as consisting of physical and non-physical *objects*, *properties* associated with these objects, *relations* between these objects, and *events* in which these objects participate. The properties and relations associated with objects will be called *features*. Features may be static or dynamic. Due to the potentially dynamic nature of a feature, that is, its ability to change values through time, a *fluent* is associated with each feature. It is this fluent that we want to model. Both features and events are extended in the temporal dimension, which means that they have value at specific time-points or over certain time-intervals. The main difference between events and features is that features have a specific value on each time-point of the interval it is defined on, while an event defined on the same interval only has a single value for the whole interval.

Example objects are *the UAV*, *the car being tracked* and *the entity observed by the camera*. Some examples of features are the *velocity* of an object, the *road segment* of a vehicle, and the

distance between two car objects. The *change in road segments* of a car, the *observation of the velocity* of the helicopter, and the *overtake* of a car are examples of events.

3.2.1.2 Representational Structures

A *representational structure* is a data structure with a well-defined interface that is able to represent knowledge about an entity and answer questions about it. Each entity has at least one type of representational structure. In DyKnow there are five different representational structures: fluent generators, fluent streams, functions, objects, and classes. Fluent generators and fluent streams are the representational structures for features. Objects and classes links are the representational structures for objects and domains of objects, and function is the representational structure for functions. The interface is dependent both on the represented entity and the representational structure. For example, the interface to the representational structure of a fluent generator has a method to ask for the value at any time-point while the fluent stream representational structure has a method to ask for the value at or before a certain time-point. The next section will describe the details of the fluent stream, fluent generator, and functions representational structures. The other representational structures will be left out in this document since they are not used by the implementation of the ESGM.

The purpose of a representational structure is to represent some entity that we want to model. There are both primitive and derived representations of all types. A primitive representational structures are based on external observations of the entity which it is representing. A derived structure on the other hand is computed from other representations, which means that we can create complex, recursive representational structures.

An example of a representational structure is a fluent stream representation of a feature. A fluent stream is a partial representation of the fluent associated with the feature, where the sequence of observations of the value of the fluent at specific time-points, called *samples*, are seen as a stream of values of the fluent. This stream may have certain properties such as maximum size, i.e. number of samples in the stream, and restrictions on the relations between the samples in the stream, such as maximum delay between two samples. A representational structure is specified by a policy. A policy could for example specify a limit on the delays of all the samples in a fluent stream. This means that the representational structure has to satisfy the constraints of the policy. There are usually many representational structures that satisfy the policy, but all of them share the properties declared in the policy. In the example all representational structures which only contain samples with a delay lower than the threshold satisfy the policy. The actual implementation of the representational structure will only maintain one of these possible structures.

3.2.1.3 Locations

A *location* is a knowledge producer and therefore a host for representational structures. Each location is seen as a separate node in a distributed system. To be a location a CORBA object has to implement the location interface for creating representational structures from policies. A location only has to support a subset of the possible policies and many locations may support the same policies.

Definition 3.2.1 (location) *A location is a knowledge producer that is a host of representational structures. It provides an interface to maintain and interact with the hosted structures.*

All representations must be hosted by some location. The reason for including locations is that we are working in a distributed system and for example approximated fluent representations of the same feature can be located in several different places in the architecture. By representing these different places with locations we make it possible to model and reason about them. Different locations might give the representational structures hosted different properties, such as storage capacity, minimum delays and interaction possibilities, specified by the policies the location supports.

A sensor is an example of a possible location where fluent approximations associated with a particular sensor may have access limitations such as only providing access to the latest value of the fluent. For example, a sensor may place an upper or lower bound on legal sampling capability or the minimum delay between measurement and availability of the observation. A database is another example of a location where one may be allowed complete access to the history of a fluent approximation.

The same feature may have fluent approximations at several locations. The reason is that the locations may have different delays and support different access methods. This is useful when processing knowledge since various functionalities have different requirements on type, quality, density of data, etc. Feedback controllers have much different requirements on fluent approximations than inference mechanisms do. For instance, the position of an autonomous agent may be accessed directly from a fluent stream from a virtual sensor or from a fluent stream in a database. The difference being that the values from the sensor will have shorter delays but only allow access to the latest value, while the database can provide the cached history of values useful for predicting future values.

3.2.1.4 Policies

A *policy* is a declarative representation of a representational structure. Policies are typed and each kind of representational structure has its own type of policy with specific definitions and constraints.

A policy has three parts: the location, the content definition, and the structural constraints. The location is the host of the representational structure. At first glance, this may appear as an ontological level error because software architectures are intended to implement declarative specifications of entities in our ontology. On the other hand, it is useful to use the representational domain which the architecture implements as a domain of discourse for an autonomous agent if one is interested in reasoning about limited reflective capability as to source, quantity and quality of data flowing through a system and dynamically specifying its use by various functionalities to achieve tasks.

The content definition specifies how the content of the representation is derived. The content can either be primitive, i.e. produced by a source outside DyKnow, or derived, i.e. produced by DyKnow. If it is derived then the policy specifies what other representational structures it is dependent on. The structural constraints restricts the representational structure in order to guarantee certain properties of the structure itself. Since the representational structure is implemented as a data structure in a computer it has to be finite, while the representational content could very well be infinite. Therefore the structure can be seen as a selection of or filter on the content. The structure also has quality of service constraints specifying some aspects of the implementation of the representational structure.

3.2.1.5 Descriptors

A *descriptor* is a reference to a representational structure and is the key concept in the knowledge mediation functionality of DyKnow. A descriptor acts as a proxy by providing an interface to the underlying representational structure. The meaning of a descriptor is defined by the policy which it is currently associated with. A descriptor is typed, where each kind of entity has its own descriptor type. The purpose of separating references from representational structures is to simplify the specification of recursive policies and to allow different representational structures to be referred to by the same descriptor at different times. This flexibility is needed in applications where the knowledge representation is dependent on changing circumstances, such as context dependency or limited resources. A descriptor is used to interact with the representational structures they denote either to declare more abstract policies and thereby representational structures or by components external to DyKnow.

3.3 Representing Features

The most common and important of the three representational entities object, feature and event is the feature. If a knowledge processing middleware were to support only one entity then it should be features. The reason is that most of the knowledge required by an agent is in this form. An agent needs to know its position, its speed, the position of obstacles in its path and so on. It doesn't have to reason about the objects which are in its path, what class of object it is or whether two obsereved obstacles are in fact the same obstacle, it is enough to know the position of those obstacles, i.e. the position feature of these objects.

A *feature* is some property of the world, it can be a property of an object (like the speed of a car), a relation between objects (the distance between two cars) or a general property about the world (the number of cars in the world). Due to the potentially dynamic nature of a feature, that is, its ability to change values through time, a *fluent* is associated with each feature. A fluent is a total function from time to value and represents the value of the feature at each time-point. A feature has exactly one fluent which is its true value over time. This actual fluent, which may be continuous, will almost never be known by an agent due to uncertain and incomplete information. There are for example inherent limitations in the sensors and in the processing which further affects the uncertainty. Instead we have to create approximations of this fluent. It is also important to realize that there is not a single best approximation which can be used in all situations, but rather what is an appropriate approximation will depend on the current task. When executing some tasks it is more important to have the latest information as soon as possible, even though it will be more uncertain and might contain occassional errors. This is usually the case for tasks continuously controlling some piece of equipment, since they make corrections all the time it doesn't matter if a correction once in a while is wrong. In other tasks, for example those involving collecting information which is to be used at a later date, it is more important that the information is as accurate as possible. In this case it is better to use potentially computationally expensive algorithms which correlates the value with other observations in order to estimate the best possible value. Some applications might even have to switch between tasks having different characteristics depending on the current situation. Therefore it is important to be able to configure the approximation and also to change it during execution. We are not interested in a static system.

A very common use of features in a system is to only consider the most current value of a feature in order to make a decision. For example, most control applications, such as a camera controller trying to track a target, needs to evaluate whether to turn some actuator on or off based

on the current value of a feature acting as the reference signal. Other applications need to update some kind of state after each new observation, for example a Kalman filter trying to estimate the position of the agent based on uncertain samples of the current position. All of these applications only need to process each sampling of the current value of a feature once. These are typical stream uses of a feature, since the sequence of updates or readings of the current value of a feature can be seen as a stream of "current values" flowing past the controller or filter. Each time a new value becomes available the application wants to be notified about the new value in order to be able to react to it. What the application would like to do is to set up a subscription to updates of the feature and associate a function to be called each time an update is available. Then the application can passively wait until notified when a new value becomes available. In DyKnow this feature usage is handled by a representational structure called a *fluent stream*. The fluent stream views the updates to the feature as a stream of timestamped values. Each update is made available on the stream for the application to react on. In order to configure the stream many different constraints can be placed on it. For example it is possible to specify a sampling period of how often the application needs new updates or to filter out certain values which are not of interest. These constraints are a declarative specification of the desired properties of the stream. DyKnow will do what it can in order to achieve these properties, but that may not always be possible. If it is not possible then the application will be notified about the violation. An application may subscribe to more than one fluent stream or specify new streams which are derived by transforming many streams into a single new stream.

Another common use of features is to ask queries about the value of a feature at particular time-points. For example, a vision system might get a timestamped picture as input and would like to know what was the position and orientation of the camera at that time-point in order to compute the location in the world of the objects found in the image. In DyKnow this feature use is handled by a representational structure called a *fluent generator*. A fluent generator is a procedure which can produce, on-demand, an approximated value of a feature for any time-point. This means it implicitly represents the value of a feature at all time-points, i.e. the fluent of a feature. To derive a value at any time-point the fluent generator will have to estimate or predict the value based on the information available to it when the request was made. Since this information is also time-dependent it means that a fluent generator may produce different values for the same time-point depending on when the value was requested. For those time-points where it is not able to estimte or predicate a suitable value it may return the special value **NoValue** to indicate this. An example would be the value at all time-points before the creation of a feature. In most applications the agent will not have sensors which produce fluent generator representations directly, instead most sensors produce a stream of samples from a feature represented by a fluent stream.

It is possible to convert between the two representational structures. A fluent stream can be transformed into a fluent generator by providing an approximation strategy, which estimates the value at those time-points where no explicit sample exists in the stream. To derive a fluent stream from a fluent generator we could for example sample the generated fluent at particular time-points to get the samples in the stream.

To summarize we have two different representational structures, fluent generators and fluent streams, for the representational entity feature. Each of these structures have different properties and different query interfaces, which are useful in different applications. The purpose of this section is to describe these two representational structures for representing features in detail.

3.3.1 Samples

The value of a feature, i.e. a property or relation, at a particular time-point is called a *sample*. It can for example be an observation or an estimation of the feature at a specific time-point. A sample consists of a value and three timestamps, the valid time, the create time and the query time. This section will explain the meaning of a sample and its components.

The value of a sample can be anything, including numbers, strings, complex structures, images, probability distributions, etc. There is also the special value **NoValue** which is used to represent that there is no value. In order to represent for example states a value can also be a structured value, which is a tuple of values. A structured value can include other structured values as well.

Definition 3.3.1 (primitive value) *A primitive value is an integer, a real, a boolean, a string, a time-point or any other primitive data type that is required by an application.*

Definition 3.3.2 (structured value) *A structured value is a tuple of values.*

Definition 3.3.3 (value) *A value is either the special value **NoValue**, a primitive value, or a structured value.*

DyKnow uses three different timestamps when talking about the time of a value. The first timestamp is the *valid time* which is the time-point when the value of the sample is assumed to be true. For example, if at time-point t we read the current position of the agent from a sensor then this sample would have the valid time t. If this position is compared to previous positions and maybe also to the current speed of the agent and a new value is estimated then the valid time would still be t even though the processing itself took some time. The definition is given below.

Definition 3.3.4 (valid time) *The valid time of a sample is the time when the value is assumed to be true.*

The concept of valid time is the same as the one used in temporal databases, where "[t]he valid time of a fact is the time when the fact is true in the modeled reality" [1].

The second timestamp is the *create time* which is the time-point when the particular value of the sample was created. One reason to introduce the create time is to be able to separate two samples from the same feature with the same valid time. The example above is a situation where this is necessary. In that case the first sample would contain the position as read from a sensor and the second sample would contain the position after it has been updated based on previous positions. In order to know which of these two samples are the most recent we need the create time, as defined in Definition 3.3.5

Definition 3.3.5 (create time) *The create time of a sample is the time when the value of the sample was created.*

There are at least three reasons for separating the valid and create times. First, several estimations of the value at a specific time-point can be made for the same feature. The different estimations would have the same valid time but different create times, as in the example above. Second, it allows us to model delays in the availability of the value. The delay would be the difference between the create time and the valid time, the greater the difference the greater the delay. The delay could for example be caused by processing of the value. If the value is not delayed then

the create and valid time will be the same. Third, it allows us to detect when a value is predicted. If the create time is less than the valid time then we know that the value must be predicted, since it can not have been observed already since the timestamps comes from the same time-line. If the create time is greater than or equal to the valid time then we can not determine if the value is an observation or an approximation. The age of the value can be determined by comparing the create time to the current time. The larger the difference the older the value.

The create time is similar to the transaction time in temporal databases [1; 3]. The transaction time is the time when the value is committed to the database, i.e. when the value is available in the database. In the same way the create time is the time-point when the value is created and is therefore available in the system.

To take another example, if a picture is taken at time-point t then the valid time of the picture is t. In the case of reading a sensor the valid time will almost always be the same as the create time, which is the current time when the value was read from the sensor. If then a blob is extracted from the picture the valid time of the blob will still be t but the create time will be time when the extraction was finished. When we have a computation that uses several inputs it is not always obvious what the valid time is. For example, if we have a Kalman filter that estimates the current position of the helicopter then the valid time of the output will always be the same as the create time since the value is the currently estimated value. If the inputs to a computation have the same valid times then most likely the valid time of the output should be the same. For example, if we have a computation that calculates the distance between two positions and the two inputs have the same valid times then the distance sample also has the same valid time. But, if the positions have different valid times, then what should the valid time of the result be? One option is not to compute any value, unless the valid times are the same, or to do some kind of estimation and then the valid time of the output should be the same as the time-point the estimation was made for.

The third timestamp is the *query time*, which is the time-point when the sample was created. The reason it is called the query time is because it is the time-point when the query was asked to which the sample is the answer. For example, if a user at time-point t_1 asks a query what is the value of a feature at time-point t_2 then the query time of the sample will be t_1 and the valid time t_2. What the create time is will depend on when the value was created. It is possible that a new value is created in order to answer the query, in that case the create time will be equal to the query time. Otherwise it will be the create time of the current value with valid time t_2. Since the times are from the same timeline the query time can never be before the create time. The query time is not strictly needed but can be useful because queries to a feature may return different values for the same valid time since each answer represents the best available approximation at the time of the query.

Definition 3.3.6 (query time) *The query time of a sample is the time when the sample was created.*

One use of the query time is to determine if the value is the best possible approximation we will get given that we know the maximum delay to produce the value of a feature at a particular time-point. If the maximum delay is 500ms and the valid time is t then if the query time is at least t+500ms then we know that this is the correct value at time-point t since otherwise the sample would be delayed more than the maximum delay. This can be relaxed so that the larger the difference between the query time and the valid time the more likely it is that this value is the best approximation the system can provide. The query time can also be used to determine the age of a sample by comparing it to the current time. The larger the difference the older the sample.

To summarize, a sample of a feature f with the value v, the valid time t_v, the create time t_c, and the query time t_q should be interpreted as: According to the source the value of f is v at t_v using the available information at t_q and the value was created at t_c. This means that the sample does not consider observations about time-point t_v made after time-point t_c or information available at the source after t_q. This description is expressed in the following definition.

Definition 3.3.7 (sample) *A sample is the value of a feature at a particular time-point, called the valid time of the sample. The value is observed or otherwise estimated at a time-point called the create time. The sample itself is created as the result of a query which was asked at a time-point called the query time.*

In certain cases it is necessary to represent that there is no sample which is the result of some operation. For example, since a fluent stream may be empty an operation returning the latest sample in the stream must be able to return something which indicates that there is no latest sample. To represent that there is no sample which is the answer to a query the special sample **NoSample** will be used.

3.3.2 Functions

A *function* is a representational structure which represents a computation, also known as a *computational unit*. A function takes a number of samples as input and compute a new sample as a result. A function can be used to derive either a computed fluent stream or a computed fluent generator. The inputs are connected to either fluent streams or fluent generators and then the function is called at the appropriate times, which is dependent on the policy used to define the derived representational structure. There are actually two types of functions, those that compute a value from the inputs without taking time into account and those that compute a value at a particular time-point. All functions must support the timeless compute functionality, but it is optional to support the compute at functionality. Those functions that support compute at a specific time-point are called *temporal functions*. There are at least two situations where temporal functions are needed. First, they are needed if the function is used to derive a computed fluent generator when the input comes from fluent streams and not from fluent generators. In this case there is no guarantee what time-points the samples in the fluent streams are from but the fluent generator must still compute the value at any time-point. Second, the functionality is needed if the function is used to derive a sampled fluent stream when the input comes from fluent streams with incompatible sample periods. In this case it might not be possible to derive inputs to the function with the requested time-points, instead the function must do the estimation itself.

Definition 3.3.8 (function) *A function is a representational structure for computations on features. A function takes a number of samples as input and computes a sample as output. A temporal function also takes a time-point as input and compute the value at this specific time-point based on the sample inputs.*

Examples of functions are filters, such as Kalman filters, and procedures for extracting information from different sources such as data fusion algorithms. An example of a data fusion algorithm is the colocate function which takes a position in an image, the position of the helicopter and the pointing direction of the camera and computes the position in world coordinates of the position in the image.

3.3.3 Fluent Streams

A *fluent stream* is a representational structure where a feature, i.e. a property or relation, is represented by a stream of samples. Each sample represents the value of the feature at a specific time-point, which is the valid time of the sample. A sample can either come from an observation of the feature or a computation which results in an estimation of the value at that time-point. If we order these samples by the time they are available we get a stream of samples representing the value of the feature over time, that is, an approximation of its fluent. The time-point when a sample is made available or added to a fluent stream is called the add time.

Definition 3.3.9 (fluent stream) *A fluent stream is a stream, i.e. an ordered sequence, of samples used as a representational structure for a feature. The samples are totally ordered by the add time, which is the time a sample was added to the stream.*

To be able to totally order the samples in a fluent stream we impose a requirement that the add time is unique, i.e. at most one sample may be added to a fluent stream at any given time-point.

Previously we discussed the similarity between the create time of a sample and the transaction time used in temporal databases. In fact, the add time is even closer to the concept of transaction time since it is the time when a sample was added or committed to a fluent stream. This means that a sample could have many "transaction times" since it could be added to several fluent streams. This is not as strange as it sounds. The reason is that the transaction time is relative to a database, and each fluent stream could be seen as a separate database. Therefore each of these databases will have different transaction times, in this case called add times, for the same sample. The difference between the add time and the create time is that the create time should be seen as the first transaction time in the system, i.e. the first time the value was committed to any database, which in DyKnow means any representational structure at any location. It is also possible to make the interpretation of the create time as the first time when the value could be available, i.e. it would be a property of the modelled world not of the representation of the model. We do not use this interpretation since it is less well defined and hard to obtain from the modelled reality.

The reason we use the concept of a stream instead of for example time-serie or ordered sequence is that it provides use with some appropriate intuitions about its properties. A stream indicates that we have something active, which is moving in some direction. In our case it is time that is moving. At any time the fluent stream will contain a finite selection of samples of a possibly infinite and continuous fluent. If we view a fluent as an infinite stream of samples, then the selection could be viewed as a moving window based on the current time over this infinite stream. The idea is to make it possible to handle and implement an infinte entity. The size of the window is not part of the definition of the fluent stream. Instead it is required that only the oldest sample is allowed to be removed from a stream and if a sample is removed then it is not added again. This disallows for the window to jump back and forth over the inifinte stream and instead keep on moving in the same direction, which is towards the future.

Even though the stream is ordered by the add time of the sample, it does not mean that it is ordered by any of the other timestamps in the sample. There are special cases where the stream is ordered, for example, by the valid time, but it is not the general case. In many applications this is a desired property therefore we will introduce further constraints on fluent streams later on in order to provide streams ordered by valid time as well as other properties.

It is also possible that a fluent stream contains several samples with the same valid time. For example, assume we have a stream containing all the approximations of the value of a feature then it would be natural to have more than one sample with the same valid time because we could update

our current estimation of the value of the feature at a particular time-point as more information becomes available.

3.3.3.1 Fluent Stream Constraints

The fluent stream constraints is a structure containing six constraints: a change, a cache, a filter, a duration, a order, and a delay constraint. The individual constraints are described below.

- Change constraint: A change constraint restricts the possibility to add samples to a fluent stream. Only those samples that are different from the last sample in the stream, according to the change constraint, are allowed to be added. This is the same as constraining the relation between each pair of consecutive samples in the fluent stream. The possible change constraints are:
  - Any change: The value of each sample in the stream must be different from the value of the previous sample.
  - Any update: Either the value or one of the time stamps of each sample in the stream must be different from those of the previous sample.
  - Sample every t timeunits: The difference in the valid time between each pair of consecutive samples should be equal to the sample period t. A new sample should be added at each sample time-point.
- Cache constraint: A cache constraint restricts what samples are allowed to be in the fluent stream at the same time. The purpose of the constraint is to limit the size of a fluent stream. The possible cache constraints are:
  - All elements: A sample that is added to the fluent stream is never removed from it.
  - t timeunits: The maximum difference in the valid times between all the samples in the fluent stream is below the time limit t. When a new sample is added all samples which then become too old should be removed.
  - n elements: The number of samples in the fluent stream is at most n. When a new sample is added to a fluent stream containing n samples, the oldest sample should be removed. If n is 0 then it is the same as an all elements cache constraint.
- Filter constraint: A filter constraint is a predicate that should be satisfied for all samples in the fluent stream. Since we have no filter language of our own we rely on the filter being written in an implementation specific language. For the current implementation using notification channels the filter must be written in the extended trader constraint language.
- Duration constraint: A duration constraint restricts the allowed valid times. The valid time of each sample has to be within the duration interval. To represent infinite durations an end time of 0 is interpreted as infinity. To say that a fluent stream should start now, a start time of 0 is interpreted as the current time when the fluent stream is created.
- Order constraint: An order constraint restricts the order of the samples in a fluent stream according to the valid time. This is the same as restricting the relation between the valid times of two consecutive samples in the fluent stream. The possible order constraints are:
  - any order: No restriction on the samples at all.

  - monotone order: Each sample must have a valid time not less than the sample before it.
  - strict monotone order: Each sample must have a valid time higher than the sample before it.

- Delay constraint: A delay constraint restricts the difference between the valid time and the time the sample is added to the fluent stream. There are two purposes of the delay constraint. First, it is used to constrain the maximum difference between the valid time and the time the sample is added to the fluent stream, which is also the time when the sample is sent to all the subscribers of the fluent stream. Second, it is used to wait for delayed samples in order to satisfy the other constraints on the fluent stream. For example, if a fluent stream has a monotone order constraint and two samples are received by the fluent stream in the wrong order then we have a policy violation. But, if we allow a delay then we can use the delay to wait for the samples arriving out of order. The effect is that as long as the receive time is within the allowed delay the samples in the fluent stream are sorted according to valid time.

3.3.3.2 Fluent Stream Policies

The fluent stream policies can be described in two steps. First the dependency on other representations are described and then the constraints placed on the fluent stream.

- fstream(l, vt, c): Create a primitive fluent stream representation at the location l storing values with the value type vt. The fluent stream will satisfy the fluent stream constraints c.
- fstream(l, named(n), c): Create an instance of the primitive fluent stream named n at the location l. A named primitive fluent stream is a representation that is connected to an external source, such as a sensor, to get the values in the fluent stream. The fluent stream will satisfy the fluent stream constraints c.
- fstream(l, s, c): Create a fluent stream representation at the location l from the fluent generator representation denoted by the descriptor s. The fluent stream will satisfy the fluent stream constraints c.
- fstream(l, $f(s_1, \ldots, s_n)$, c): Create a fluent stream representation at the location l computed from the function denoted by the descriptor f. The inputs to the function are denoted by the descriptors $s_1, \ldots, s_n$. The fluent stream will satisfy the fluent stream constraints c.
- fstream(l, sync($s_1, \ldots, s_n$), c): Create a fluent stream representation at the location l with synchronized states. Each state contains an element from each of the input streams denoted by the descriptors $s_1, \ldots, s_n$. All the elements in each state are valid at the same time, which is the valid time of the state. The fluent stream will satisfy the fluent stream constraints c.
- fstream(l, merge($s_1, \ldots, s_n$), c): Create a fluent stream representation at the location l merging the input streams denoted by the descriptors $s_1, \ldots, s_n$ to a single fluent stream. The fluent stream will satisfy the fluent stream constraints c.

3.3.4 Fluent Generators

As mentioned in the introduction there are two main usages of features, either to get the current value or to get the value of the feature at any time-point. The first case is best handled with a

fluent stream as described in the previous section, while a fluent generator is the best choice in the second case. A fluent generator is a representational structure which can produce an estimation of the fluent of a feature, i.e. a total function from time to value representing the value of the feature at all time-points. This can not be done with a fluent stream since it is only a sequence of samples. The sequence may be a partial function from time to value, but it is not always the case since it may contain several updates of the value at the same time-point. The fluent generator, on the other hand, produce an estimation of the complete fluent at every time-point. The fluents generated does not have to be the same, but will most often depend on when the fluent was generated. There are at least reasons for this related to the non-monotonic nature of knowledge and the bounded amount of resources available. First, the fluent generator may use all the available information about the feature when generating the fluent and this information will change over time as more information becomes available. For example, a fluent generator will always have to do predictions of values in the future, but as time progress more and more time-points are in the past and may have information based on observations. Second, the fluent generator is still a finite representational structure which means that it may have to forget or discard old information. In systems which either does not generate very many observations or have a lot of memory it might be possible to store the complete history of a feature, but this is not the normal case. Since the fluent generated may change over time the fluent generator can be seen as a function from time to fluent. The fluent generated at a time-point is the best estimation of the fluent that the fluent generator can produce.

Definition 3.3.10 (fluent generator) *A fluent generator is a function from time to fluents used as a representational structure for a feature.*

The purpose of the fluent generator is to represent an infinite fluent with a finit representational structure by providing a procedure to compute, on demand, the value of the fluent at a particular time-point. The idea is to allow an application to query the implicit fluent for the required information, which will then be produced by the fluent generator.

From a fluent stream it is possible to derive a fluent generator by provinding a method for approximating the value at each time-point based on the content of the fluent stream. The approximation needs to consider the possibilities of zero, one, or more samples in the fluent for each time-point. How this approximation is done is part of the specification of the fluent generator.

It is also possible to derive a fluent stream from a fluent generator. One way is to sample the fluent generator at regular time-points to derive a sampled fluent stream. It is also possible to do more advanced analyses of the fluent generator in order to derive all the changes in the generated fluent to derive a fluent stream which contain only the changes in the fluent.

3.3.4.1 Fluent Generator Constraints

The fluent generator constraints is a structure containing two constraints: a value approximation constraint and a merge constraint. The individual constraints are described below.

- Value approximation constraint: Approximate the value at time-point t if no explicit sample exists at that time-point. The possible value approximation constraints are:
  - No approximation: A value not available exception will be thrown if trying to approximate the value.
  - Closest value approximation: The value at time-point t is approximated to the value with the valid time which is the closest to t. It can be either before or after the time-

point t. If there are no values either before or after t then a value not available exception is thrown.

- Most recent value approximation: The value at time-point t is approximated to the value with the valid time which is the closest before t. If there is no value before t then a value not available exception is thrown.
- Default value approximation v: The value at time-point t is approximated by a default value v.
- Linear interpolation value approximation: The value at time-point t is approximated using a linear interpolation between the value which is the closest before and after t. If there is no value either before or after t then a value not available exception is thrown.

- Merge constraint: Approximate the value at time-point t if more than one value exists at that time-point. The possible merge constraints are:
  - Keep first: Keep the sample with the lowest query time.
  - Keep last: Keep the sample with the highest query time.
  - Average: Take the average of all the sample from time-point t.
  - Median: Take the median of all the sample from time-point t.

3.3.4.2 Fluent Generator Policies

The fluent generator policies can be described in two steps. First the dependency on other representations are described and then the constraints placed on the fluent generator.

- feature(l, vt, c): Create a primitive fluent generator representation at the location l storing values with the value type vt. The fluent generator will satisfy the fluent generator constraints c.
- feature(l, named(n), c): Create an instance of the primitive fluent generator named n at the location l. A named primitive fluent generator is a representation that is connected to an external source, such as a sensor, to get the values in the fluent generator. The fluent generator will satisfy the fluent generator constraints c.
- feature(l, s, c): Create a fluent generator representation at the location l from the fluent stream representation denoted by the descriptor s. The fluent generator will satisfy the fluent generator constraints c.
- feature(l, $f(s_1, \ldots, s_n)$): Create a fluent generator representation at the location l computed from the function denoted by the descriptor f. The inputs to the function are denoted by the descriptors $s_1, \ldots, s_n$.
- feature(l, state($s_1, \ldots, s_n$)): Create a fluent generator representation at the location l with states. Each state contains an element from each of the input fluent generators denoted by the descriptors $s_1, \ldots, s_n$. All the elements in each state are valid at the same time, which is the valid time of the state.

Chapter 4

Implementing the ESGM Using DyKnow

The entity structure generation module is actually a wrapper around two modules which are part of DyKnow, the *dynamic object repository*, also known as the *DOR*, and the *symbol manager*. When the ESGM is requested to create an entity trajectory structure it will create a suitable policy and supply this to the symbol manager which will make sure that an appropriate representational structure is created in the DOR. Apart from these main components, which are implemented as CORBA servers, there are a number of auxiliary servers providing necessary services. The servers used in this DyKnow setup are shown in Figure 4.1. The IOR server is an alternative to the CORBA nameservice, that we use mainly for historical reasons, which associates names with CORBA objects. By using common names applications can get hold of the necessary servers in the distributed system by asking the IOR server for the reference. All the servers used in this example are registered in the IOR server. The time server keeps track of the current time in the system and provides an alarm functionality to create timeouts with regular time intervals.

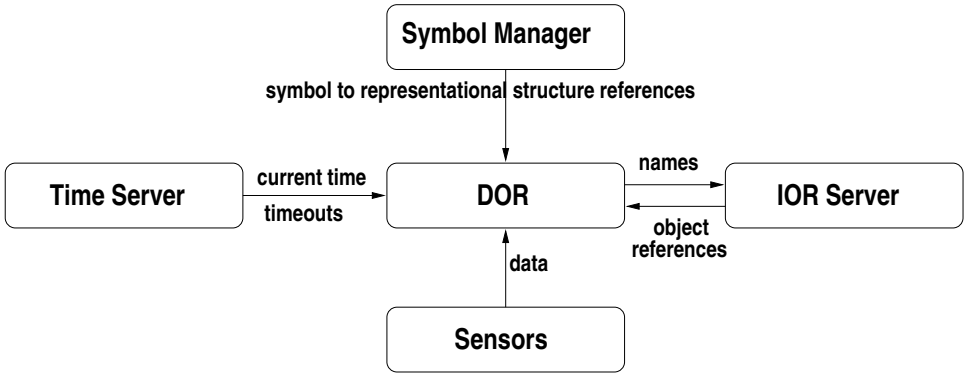

Figure 4.1: An overview of the underlying DyKnow components that the ESGM interacts with.

4.1 Integrating Data Sources

When a new data source is added to the ESGM a new named primitive fluent stream is created in DyKnow.

4.2 Integrating Computational Units

When a new computational unit is added to the ESGM it is also added to the DOR, which is a computational unit manager, and a function policy is created and associated with a descriptor having the same name as the computational unit.

4.3 Implementing Entity Structures

Entity structures are implemented as samples in DyKnow. Each entity structure is a sample where the value of the sample is the name and attribute/value pairs. The valid and create time-stamps of the entity structure are taken directory from the sample, but the query time is not used.

4.4 Implementing Entity Trajectory Structures

Each entity trajectory structure created by the ESGM is implemented by a fluent stream. The fluent stream is created from a policy and associated with a descriptor. The name of the descriptor is the same as the name of the trajectory. The policy used is determined by the type of trajectory created, as explained below.

4.4.1 Implementing Primitive Entity Trajectory Structures

The policy used to implement the primitive entity trajectory structure is the primitive fluent stream with the value type `ENTITY_FRAME_TYPE`, because that is the name of the value type that is used to store the name and attribute value pairs of an entity structure. In the current implementation no constraints are placed on the fluent stream, which means that entity structures might be added in any order, with any delay and that all entity structures are stored in the cache. A reasonable set of constraints would be to use the monotone order constraint and to allow the application to set the acceptable delay and maximum cache size.

4.4.2 Implementing Sampled Entity Trajectory Structures

The policy used to implement the sampled entity trajectory structure is the named primitive fluent stream where the name is the name of the sensor. The constraints placed on the fluent stream is dependent on the parameters supplied by the application which are the start and end time, the sample period, the maximum delay, and the cache size. The sample period is translated to a sample change constraint. The start and end time to a duration constraint. The maximum delay to a delay constraint, and the cache size to a cache constraint. A monotone order constraint is also added to make sure the entity structures are received in the correct order.

4.4.3 Implementing Computed Entity Trajectory Structures

The policy used to implement the computed entity trajectory structure is the computed primitive fluent stream where the name of the function descriptor is the name of the computational unit, and the input descriptors are the names of the input trajectories. In the current implementation no constraints are placed on the fluent stream, which means that entity structures might be added in any order, with any delay and that all entity structures are stored in the cache. If the input streams are well-behaved then the output stream should be as well, unless the implementation of

the notification service starts reordering events. A reasonable set of constraints would be to use the monotone order constraint and to allow the application to set the acceptable delay and maximum cache size.

Part II

Tutorial

Chapter 5

Introduction

This tutorial gives a practical introduction to using the entity structure generation module (ESGM) centered around a concrete ground robot example. The purpose is to show how to store the information produced by sensors in the ESGM as entity trajectory structures, how to create new entity trajectory structures by performing continuous computations on several input trajectories, and how to use these trajectories in an application. The tutorial is self-contained and should be enough to get started using the ESGM.

Consider the following scenario. A wheeled ground robot is moving around in an in-door environment looking for objects to inspect. The robot is equipped with a camera. In order to find objects of interest in the pictures taken by the camera there is an image processing algorithm called BlobDetector for finding entities called *blobs* based on their shape. The output is a trajectory of blob sequences. To filter out those blobs that are not interesting and select the current object of interest a filter called ObjectSelector is used. The output of the filter is a trajectory of blob entity structures. When the robot has found an object of interest it should approach it to have a closer look. To compute the direction to travel in there is a function called ComputeDirection which takes the object of interest, represented by a blob entity structure, and the current robot position, represented by a *robot state* entity structure, as input and produces a *driving direction* entity structure with the desired direction and speed to travel. An overview of the data flow in the application is shown in Figure 5.1.

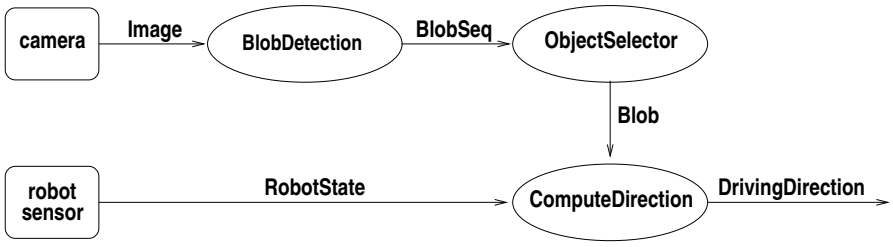

Figure 5.1: An overview of the data flow in the tutorial application. The boxes are sensors, the ellipses are computational units, and the lines are entity trajectory structures labeled with the name of its entity structure type.

All the code in the tutorial is written in C++ and is available in the supplementary file called `esgm_tutorial.cc`. The file contains the implementations of the sensor interfaces, the computational units, and the main program.

The tutorial is structured around the example application as follows. First, Chapter 6 gives the necessary information to get started and run the entity structure generation module and access it in the application. Then Chapter 7 describes the entity structures used in the tutorial and how to add them to the ESGM. Next, Chapter 8 show how to integrate the two sensors and make their information available through the ESGM. Then, Chapter 9 shows how to create the derived entity trajectory structures from feature extraction and filtering using computational units. Chapter 10 shows how to interact with the ESGM to retrieve existing entity trajectory structures and either query them or subscribe to their content as it becomes available. The complete source code of the tutorial is available in Chapter III.

Chapter 6

Getting Started

This chapter describes how to install and run the ESGM.

6.1 Installing the ESGM

- Check out the MACS code from Gibson.
- Follow the instructions in the README file in the LiU/dyknow directory on how to download the DyKnow binaries and compile the ESGM.

6.2 Running the ESGM

To run the ESGM you only have to start the program `esgm`. It is configured to start all the servers which are required by the ESGM by default. When the text `''(ESGM) ORB running...''` is printed on the screen the ESGM is up and running.

6.3 Retrieving the ESGM

Before the ESGM can be used in an application a reference to the actual ESGM CORBA server has to be retrieved. This can be done using the helper function `get_object_from_iorserver` which takes a pointer to an `IorServerProxy`, the name of the server, and a variable to store the reference in. The following code retrieves the default IorServer and retrieves a reference to the ESGM stored under the name `ESGM_NAME`.

```
Witas::Util::IorServerProxy ior_server;
Esgm_var esgm;
get_object_from_iorserver(&ior_server, ESGM_NAME, esgm);
```

If the function is successfull the variable *esgm* can now be used to call the methods in the ESGM interface.

Chapter 7

Entity Structures

To represent entities there is a data type called `EntityStructure` which is part of the ESGM implementation. The data type consists of a name, two time-stamps, and a sequence of attribute value pairs. The time-stamps called create time and valid time basically represents the time-point when the value was created and the time-point when the value is valid. They are explained in detail in section 2.4. The `EntityStructure` is a general data type which can represent any entity.

```
struct EntityStructure {
  string name;
  Time valid_time;
  Time create_time;
  AttributeValuePairSeq attributes;
};
```

7.1 Entity Structure Types

Each entity structure is an instance of an entity structure type. The entity structure type is represented by the data type `EntityStructureType` which consists of a name and a sequence of attributes. In order to be an instance of the entity structure type the entity structure must have values for all the attributes defined in the type. Before an entity structure type can be used to create instances it must be defined. How this is done is explained in Section 7.3.

```
struct EntityStructureType {
  string name;
  AttributeSeq attributes;
};
```

The tutorial uses five entity structure types: image, blob, blob sequence, robot state and driving direction. Each of them is described below.

7.1.0.1 Image

The Image entity structure type represents a single image taken by the camera. It has the following attributes:

- width : integer; the width of the image,

- height : integer; the height of the image,
- format : integer; the format of the image represented by a number,
- image_data : octet sequence; the data of the image represented according to the format.

7.1.0.2 Blob

The Blob entity structure type represents a single blob found in an image. It has the following attributes:

- x : integer; the x-position within the image,
- y : integer; the y-position within the image,
- size : integer; the size in pixels.

7.1.0.3 BlobSeq

The BlobSeq entity structure type represents a sequence of blobs. It has the following attributes:

- items : entity frame sequence; the sequence of blobs.

7.1.0.4 RobotState

The RobotState entity structure type represents the state of the robot. It has the following attributes:

- x : double; the x-coordinate of the position of the robot,
- y : double; the y-coordinate of the position of the robot,
- dir : double; the direction of the robot in degrees.

7.1.0.5 DrivingDirection

The DrivingDirection entity structure type represents the desired driving direction of the robot. It has the following attributes:

- dir : double; the desired direction,
- speed : double; the desired speed.

7.2 Entity Structure Wrappers

Since the entity structure is very general it means that it is quite clumsy to use directly. In order to make the interactions with it more user friendly there is a utility class called `EntityStructureWrapper`. Since this is also a general interface it is still not as user friendly as a struct for each entity structure type would be. Therefore it is good idea to specialize the `EntityStructureWrapper` for each entity structure type to come as close as possible. As part of the tutorial code there are the following specialized wrappers:

- `ImageWrapper`,
- `BlobWrapper`,
- `RobotStateWrapper`, and
- `DrivingDirectionWrapper`.

There is also the general `SeqWrapper` which encapsulates any entity structure which represents a sequence of entity structures. These wrappers makes the entity structure almost as user friendly as a simple struct would be. The difference is that instead of treating the members of the struct as attributes they are treated as methods. For example to get the size of a blob an application would call the method `size`() which returns the size. To set the size it would call the method `size`(*new_size*) with the desired new size.

7.3 Adding Entity Structure Types

Before an entity structure type can be used it has to be declared to the ESGM. The following code declares the Blob entity structure type:

```
EntityStructureType blob_type;
blob_type.name = CORBA::string_dup("BlobType");
blob_type.attributes.length(3);
blob_type.attributes[0].name = CORBA::string_dup("x");
blob_type.attributes[0].type = LONG_TYPE;
blob_type.attributes[1].name = CORBA::string_dup("y");
blob_type.attributes[1].type = LONG_TYPE;
blob_type.attributes[2].name = CORBA::string_dup("size");
blob_type.attributes[2].type = LONG_TYPE;

esgm->create_entity_structure_type("blob", blob_type);
```

To make it easier to add the entity structure type the specialized entity structure wrappers have a static method called `type` which returns an `EntityStructureType` which can be used to create the type. The following code declares the Image entity structure type using the `type` method:

```
esgm->create_entity_structure_type("image", ImageWrapper::type());
```

Chapter 8

Making Sensor Data Available to the ESGM

Now that the entity structure types are defined and user friendly wrappers are available we can start entering data into the ESGM. The data sources we have in this application are the camera and the robot state extraction mechanism. In order to show the two different ways of integrating data sources we will make the ESGM pull images from the camera sensor and make the robot sensor push data to the ESGM when it has a new robot state available.

8.1 Creating a Primitive Entity Trajectory Structure for the Robot States

To integrate a sensor using the push mechanism the sensor has to create a primitive entity trajectory structure and every time it has a new entity structure ready add it to this trajectory. When the trajectory is updated any subscriber will be notified so they can react to the new sensor reading. The following code creates a primitive entity trajectory structure:

```
EntityTrajectoryStructure_var robot_state
 = esgm->create_primitive_entity_trajectory_structure("robot_state",
      "RobotState");
```

The variable *robot_state* is a reference to an `EntityTrajectoryStructure` object and can be used to interact with the entity trajectory structure. It can for example be used to add a new entity structure to the trajectory, as showed in the following code:

```
RobotStateWrapper rs("robot1", 1, 2, 3);
robot_state->create_at_now(rs.name, rs.attributes);
```

The tutorial code creates a thread which adds a new entity structure every 100 milli-seconds to the robot state entity trajectory.

8.2 Creating a Sensor Interface to the Camera

To integrate a sensor using the pull mechanism the sensor has to implement an interface called `SensorInterface`. The interface has only one method `get_current_value`(*qtime*) which should return a pointer to a new sample (which is the data type used by DyKnow to represent

a single observation). The argument *qtime* is the query time which is the time-point when the query to the sensor interface was asked. This time-stamp can be used to return the entity structure closest to this time-point. But since it should be more or less equal to the current time it is enough to return the most current entity structure.

The following code implements the camera sensor interface. Instead of extracting a picture from the frame grabber the code creates a dummy image. Both the create and valid time-stamps of the entity structure are set to the current time which is approximated to be the time-point when the query was asked.

```
class CameraSensorInterface
  : public virtual POA_Witas::Idl::Dyknow::SensorInterface
{
public:
  CameraSensorInterface() {}

  virtual Sample* get_current_value(Time qtime)
    ACE_THROW_SPEC ((CORBA::SystemException));
};

Sample*
CameraSensorInterface::get_current_value(Time qtime)
  ACE_THROW_SPEC ((CORBA::SystemException))
{
  OctetSeq img_data;
  img_data.length(640*480*3);
  for ( CORBA::ULong i = 0; i < 640*480; ++i ) {
    img_data[3*i]   = 0; // red
    img_data[3*i+1] = 0; // green
    img_data[3*i+2] = 0; // blue
  }
  ImageWrapper img("image", 640, 480, img_data, 0);
  Time now = qtime;
  return new mSample(mValue(img), now, now);
}
```

To make a sensor interface available to the ESGM the application first has to create an instance of the implementation, and then register it with the ESGM. The registration associates the sensor interface with a name and the type of the entity structures created. The following codes does that:

```
CameraSensorInterface camera_sensor_iface_servant;
SensorInterface_var
  camera_sensor_iface = camera_sensor_iface_servant._this();
esgm->add_sensor_interface("camera", image_type.c_str(),
   camera_sensor_iface.in());
```

It is important to note that this code only makes the sensor available to the ESGM for creating entity trajectory structures, no actual trajectory is created. The benefit is that the same sensor interface can be used to create many entity trajectory structures with different properties. This is the major difference compared to the push mechanism which only provides a single entity trajectory structure.

To create an entity trajectory structure with images from the camera sensor a request has to be made to the ESGM with the parameters for this particular trajectory. The parameters include the start and end time of the trajectory, the sample period i.e. how often the sensor interface should

be called in order to get a new entity structure, the maximum acceptable delay and the number of entity structures to save in the cache. The following code creates an entity trajectory structure called *images* which goes on forever, sampling the camera once every second with any delay, and caching the last 5 images:

```
EntityTrajectoryStructure_var images
 = esgm->create_entity_trajectory_structure("images",
                                            "camera",
    0, 0,
    1*TIMEUNITS_PER_SECOND,
    0, 5);
```

The variable *images* can now be used to interact with the entity structure for example to get the current value or to subscribe to images that will be added every second. The name of the entity trajectory structure can be used for example to get a new reference to it or when making it an input to a computational unit.

Chapter 9

Transforming Information in the ESGM

With the information from the sensors available through the ESGM it is now time to start transforming this information. The main concept used to implement these transformations is the computed entity trajectory structure. The idea is to take one or more entity trajectories and transform them into a new entity trajectory by applying a *computational unit* to the input. The computational unit encapsulates how to create a new entity structure from one entity structure from each of the input trajectories. In the current implementation any synchronization necessary between these input trajectories has to be done by the computational unit, as described below. The chapter shows how to extract a sequence of blobs from an image, which is an example of a computational unit with only one input, and how to compute the driving direction based on the current state of the robot and the current object of interest, which is also an example of how to synchronize the inputs to a computational unit.

9.1 Extracting Blobs

The simplest type of computational units are those with only a single input. The blob extraction is an example of such a computational unit, since it takes an image as input and outputs the sequence of extracted blobs. The input will be Image and the output BlobSeq entity structures. To implement a computational unit the application has to implement the `CompUnit` interface. The interface consists of a single method `compute`(*inputs*, *qtime*) which should return a new entity structure or throw a `ValueNotAvailable` exception if no new entity structure is computed. The inputs contains a sequence of entity structures, one for each of the inputs, if they contain any entity structures. This means that if an empty trajectory structure is connected to a computational unit then it will not have enough inputs to do its computation. To make sure that all inputs have a value check the length of the input sequence and make sure it is correct. An example is shown in the code below. The inputs in the sequence are in the same order as the inputs are specified when a computed entity trajectory structure is created. This is not optimal, but the current implementation. To extract the entity structures from the input sequence the wrapper utilities can be used. As in the sensor interface the time-point of the query to compute the value is available in the *qtime* parameter. The following code implements the blob detection computational unit:

```
class BlobDetectionCU
  : public virtual POA_Witas::Idl::Dyknow::CompUnit
{
public:
  BlobDetectionCU() {}
```

```
  virtual Sample* compute(const SampleSeq& inputs, Time qtime)
    ACE_THROW_SPEC ((CORBA::SystemException,
     ValueNotAvailable));
};

Sample*
BlobDetectionCU::compute(const SampleSeq& inputs, Time qtime)
  ACE_THROW_SPEC ((CORBA::SystemException,
   ValueNotAvailable))
{
  try {
    if ( inputs.length() == 1 ) {
      Time now = qtime;
      ImageWrapper image(get_EntityFrame(inputs[0].val));
      // Do stuff with image
      SeqWrapper blobs("blobs");
      return new mSample(mValue(blobs), now, qtime, now);
    }
  } catch ( const CORBA::Exception& e ) {
    cerr << "BlobDetectionCU::compute threw " << e << endl;
  }
  throw ValueNotAvailable();
}
```

To make a computational unit available to the ESGM the application first has to create an instance of the implementation, and then register it with the ESGM. The registration associates the computational unit with a name, the types of the inputs and the type of the output entity structures. The following codes does that:

```
BlobDetectionCU blob_detection_cu_servant;
CompUnit_var blob_detection_cu = blob_detection_cu_servant._this();
mValueSortSeq input_types(ENTITY_FRAME_TYPE);
esgm->add_comp_unit("BlobDetection", blob_type.c_str(),
    input_types, blob_detection_cu.in());
```

It is important to note that this code only makes the computational unit available to the ESGM for creating computed entity trajectory structures, no actual trajectory is created. To create such an entity trajectory a request has to be made to the ESGM with the parameters for this particular trajectory. The parameters are the name of the trajectory, the name of the computational unit and a sequence with the names of the inputs in the same order as they should appear in the input sequence to the computational units `compute` method. The following code creates an entity trajectory structure called *blobs* by applying the computational unit *BlobDetection* to the *images* trajectory:

```
mStringSeq inputs("images");
EntityTrajectoryStructure_var blobs
 = esgm->create_computed_entity_trajectory_structure("blobs",
     "BlobDetection",
     inputs);
```

Each time a new image entity structure is available in the *images* trajectory the computational unit *BlobDetection* will be called with it as the input. If the detector finds any blobs then they will be added to the *blobs* trajectory and any subscribers to it will be notified.

The filtering of the blobs is done in the same way and is therefore not shown here. The result is an entity trajectory structure called *current_blob*.

9.2 Computing the Driving Direction

Now that we have entity trajectory structures with the robot states and the current blob of interest we can combine these two to make a decision on which is the best driving direction in order to take a closer look. The driving direction is computed by a computational unit called ComputeDrivingDirection. The only difference compared to the computational units in the previous section is that it has two inputs. This means that it has to make sure that they are synchronized unless we want to start chasing old objects or make the same turn twice. In this case it is quite simple to only compute a new driving direction when the robot state is older than the current blob. Since new robot states are provided more often than new blobs it should at most take 100 milli-seonds (which is the current sample period of the robot sensor) before the robot changes its driving direction towards the new object. This can still give undesired effects since the robot will drive towards the last known position of the object, not its current position. An alternative would be to only change driving directions after the first new robot state received after seeing a blob. The following code implements the first alternative:

```
Sample*
DrivingDirectionCU::compute(const SampleSeq& inputs, Time qtime)
  ACE_THROW_SPEC ((CORBA::SystemException,
   ValueNotAvailable))
{
  try {
    // input[0] is the robot state and input[1] the current blob
    if ( inputs.length() == 2
 and inputs[0].vtime >= inputs[1].vtime ) {
      BlobWrapper robot_state(get_EntityFrame(inputs[0].val));
      BlobWrapper blob(get_EntityFrame(inputs[1].val));
      Time now = qtime;
      // Compute driving direction
      DrivingDirectionWrapper driving_dir("driving_dir", 10, 10);
      return new mSample(mValue(driving_dir), now, qtime, now);
    }
  } catch ( const CORBA::Exception& e ) {
    cout << "DrivingDirectionCU::compute threw " << e << endl;
  }
  throw ValueNotAvailable();
}
```

An alternative to this implementation would be to have a computational unit which only subscribes to the current blob and each time a new blob is available it queries the robot state entity trajectory structure in order to find out the current position of the robot. Based on this information it could then compute the best driving direction.

As before adding the computational unit doesn't create any trajectories. The following code creates an entity trajectory structure called *driving_dir* by applying the *DrivingDirection* computational unit to the *robot_state* and *current_blob* trajectories:

```
EntityTrajectoryStructure_var driving_direction
 = esgm->create_computed_entity_trajectory_structure("driving_direction",
     "DrivingDirection",
     mStringSeq("robot_state",
     "current_blob"));
```

Chapter 10

Extracting Information from the ESGM

After integrating the sensors and processing the outputs of the sensors our application now has a collection of entity trajectory structures which are ready to be used. To extract information from the ESGM the application first have to retrieve a reference to the entity trajectory structure it would like to interface. This is described in Section 10.1. With the reference it is possible to start extracting information about that particular trajectory. There are basically two ways of interacting with a trajectory. Either query it for specific information, as described in Section 10.2, or subscribe to it in order to continually receive the new entity structures as they are added to it, as described in Section 10.3.

10.1 Retrieving the Driving Direction Entity Trajectory Structure

To retrieve an already existing entity trajectory structure a request is sent to the ESGM with the name of the desired trajectory. If the trajectory exists a reference to it is returned, otherwise a `NoSuchEntityStructure` exception is thrown. The following code retrieves the driving direction entity trajectory structure:

```
EntityTrajectoryStructure_var driving_dir
 = esgm->get_entity_trajectory_structure("driving_dir");
```

With this reference it is possible to query the trajectory or subscribe to it as described in the next two sections.

10.2 Querying the Driving Direction Entity Trajectory Structure

When we have constructed or retrieved an entity trajectory structure with the driving directions of the robot we can query it in order to find out its content. The most basic query is to get the latest entity structure in the trajectory. If the trajectory is empty a `ValueNotAvailable` exception is thrown, otherwise a copy of the latest entity structure is returned. It is important to know that the application now controls the copy and must return the memory when done with it. The following code gets the latest value and stores it in a `_var` variable which is a smart pointer which returns the memory when it goes out of scope:

```
EntityStructure_var latest = driving_dir->get_latest();
```

Since we have a trajectory of entity structures we can also ask for previous entities, providing the cache size is large enough to contain them and enough entities have been added. To retrieve

all entity structures with valid time in a certain interval the method `get_between` can be used. It takes a start and an end time and returns a sequence of all entity structures with a valid time in that range. The following code retrieves all driving directions from five seconds before to one second before the valid time of the current driving direction:

```
Time t1 = latest->valid_time - 5*TIMEUNITS_PER_SECOND;
Time t2 = latest->valid_time - 1*TIMEUNITS_PER_SECOND;
EntityStructureSeq_var interval
 = driving_dir->get_between(t1, t2);
```

One common query is to get the entity structure at a specific time-point. For example in the alternative implementation of the driving direction computation above the idea was to wait for a new blob and then retrieve the current robot state. This would be the same as requesting the robot state which is valid at the same time as the blob is valid. Since the sampling time-points of the two trajectories are not synchronized there is a very high probability that no entity structure exists for a specific time-point. If this is the case an approximation must be made. Currently there are three approximation strategies implemented. The most common strategy is to take the most recent value, i.e. the entity structure with the highest valid time which is before the desired time-point. This is called the most recent value approximation strategy. If we assume that the entity trajectory contains all the changes in the entity structure then this would be the best possible approximation. The other two approximations are to either take the entity structure with the lowest valid time which is after the desired time-point or to take the entity structure with the valid time which is closest, either before or after, the desired time-point. The following code asks for the value at time-point t with the most recent value approximation strategy:

```
ValueApproximationConstraint aprx;
aprx._d(MOST_RECENT_VALUE_APPROXIMATION);
EntityStructure_var at_t
 = driving_dir->get_at_approximating(t, aprx);
```

The same entity structure would be retrieved if the `get_closest_at_or_before` method had been used instead.

10.3 Subscribing to the Driving Direction Entity Trajectory Structure

To be notified of new driving directions for the robot, we can either poll the trajectory manually by using funtions such as `get_latest` on the trajectory, or we can set up a subscription. When subscribing the subscriber will be notified whenever a new entity structure is added to the trajectory. This means that the subscriber does not have to poll the trajectory to see if any new entity structures are available. The subscription is implemented using the CORBA notification service. To simplify the subscription, a utility class called `SubscriptionProxy` can be used. The proxy takes a callback as argument which is called each time a new entity structure is added to the trajectory. Here is an example of such a callback which only prints out each new entity structure as it arrives.

```
class SubCallback
 : public SubscriptionCallback {
public:
  SubCallback() {}

  void got_sample(const Symbol& sym, const Sample& sample)
  {
```

```
    cout << "Got " << sym << " = " << sample.val << endl;
  }
};
```

To start a subscription the application needs to create a subscription proxy and tell it to use an instance of the callback class previously created. The following code creates a subscription on the driving direction trajectory, using the callback class `SubCallback` that will be called each time a new entity structure is received. The subscription is created from the handle returned by calling the `subscribe` method on the entity trajectory structure, a symbol associated with this particular subscription and a callback object. The proxy will setup the subscription using the subscription handle, subscribe to the notification channel, and mediate the entity structures on the notification channel to the callback object. When the proxy object is destroyed the subscription will be terminated. The symbol is used to differentiate between several subscriptions to the same callback. The symbol is one of the arguments to the `got_sample` method which is called for each new entity. The symbol does not have to be the same as the name of the entity or the entity trajectory structures, it is only used by the callback to separate multiple subscriptions.

```
SubCallback callback;
SubscriptionProxy sub_proxy(driving_dir->subscribe(),
    mSymbol("driving_dir"), &callback);
```

To receive the notification sent on the event channel the ORB must be running. Since the application is only waiting for entity structures to arrive it can hand over the thread of control to the ORB by calling its `run` method.

```
orb->run();
```

Now the application will wait for entity structure to arrive on the channel and as soon as they do, the callback will be notified and can perform its work. This concludes the example application.

Part III

Appendices

```
/**
 * @file value.idl
 * Definition of the type Value.
 */


#ifndef VALUE_IDL
#define VALUE_IDL


#include "common_types.idl"
#include "symbol.idl"

#pragma prefix "ida.liu.se/witas"


module Witas {
  module Idl {
    module Dyknow {

      /**
       * A type representing the different value types which Dyknow
       * can handle.
       *
       * @todo Replace with the type code or something else generic?
       */
      enum ValueSort {
        AGG_TYPE,
        ALTITUDE_TYPE,
        ANY_VALUE_TYPE,
        BOOL_TYPE,
        COLLECTION_TYPE,
        DOUBLE_TYPE,
        DOUBLE_SEQ_TYPE,
        ENTITY_FRAME_TYPE,
        ENTITY_FRAME_SEQ_TYPE,
        LONG_TYPE,
        LONG_SEQ_TYPE,
        OCTET_SEQ_TYPE,
        POS2_TYPE,
        POS2_SEQ_TYPE,
        POSITION_ALTITUDE_TYPE,
        POSITION_ALTITUDE_ROTATION_TYPE,
        POSITION_TYPE,
        ROTATION_TYPE,
        STRING_TYPE,
        SYMBOL_SEQ_TYPE,
        SYMBOL_TYPE,
        TIME_TYPE,
        UNKNOWN_VALUE_TYPE,
        VEC_TYPE,
        VEC_SEQ_TYPE
      };
```

```
/// Representation of a sequence of value sorts
typedef sequence<ValueSort> ValueSortSeq;


/// Representation of a value in Dyknow
union Value; // forward declaration

struct Attr {
  string name;
  ValueSort type;
};
typedef sequence<Attr> AttributeSeq;

struct AttributeValuePair; // forward declaration
typedef sequence<AttributeValuePair> AttributeValuePairSeq;

struct EntityFrame {
  string name;
  AttributeValuePairSeq attributes;
};
typedef sequence<EntityFrame> EntityFrameSeq;

/// Representation of a sequence of values, in Dyknow
typedef sequence<Value> ValueSeq;


union Value switch (ValueSort) {
  case ANY_VALUE_TYPE:
    any any_val;
  case LONG_TYPE:
    long long_val;
  case DOUBLE_TYPE:
    double double_val;
  case LONG_SEQ_TYPE:
    LongSeq long_seq_val;
  case DOUBLE_SEQ_TYPE:
    DoubleSeq double_seq_val;
  case BOOL_TYPE:
    boolean bool_val;
  case STRING_TYPE:
    string string_val;
  case TIME_TYPE:
    Time time_val;
  case AGG_TYPE:
    ValueSeq agg_val;
  case COLLECTION_TYPE:
    ValueSeq collection_val;
  case VEC_TYPE:
    Vec vec_val;
  case VEC_SEQ_TYPE:
    VecSeq vec_seq_val;
  case POS2_TYPE:
    Pos2 pos2_val;
```

```
        case POS2_SEQ_TYPE:
          Pos2Seq pos2_seq_val;
        case POSITION_TYPE:
          Position position_val;
        case ALTITUDE_TYPE:
          Altitude altitude_val;
        case ROTATION_TYPE:
          Rotation rotation_val;
        case POSITION_ALTITUDE_TYPE:
          PositionAltitude pos_alt_val;
        case POSITION_ALTITUDE_ROTATION_TYPE:
          PositionAltitudeRotation par_val;
        case SYMBOL_TYPE:
          Symbol symbol_val;
        case SYMBOL_SEQ_TYPE:
          SymbolSeq symbol_seq_val;
        case OCTET_SEQ_TYPE:
          OctetSeq octet_seq_val;
        case ENTITY_FRAME_TYPE:
          EntityFrame entity_frame_val;
        case ENTITY_FRAME_SEQ_TYPE:
          EntityFrameSeq entity_frame_seq_val;
      };

      struct AttributeValuePair {
        string name;
        Value val;
      };


      /**
       * @brief The types used are incompatible.
       * @a desired_type is the type that was requested.
       * @a actual_type is the actual type available.
       */
      exception InvalidType {
        ValueSort desired_type;
        ValueSort actual_type;
      };

    };
  };
};


#endif // VALUE_IDL
```

```
/**
 * @file entity_structure.idl
 * Definition of the EntityStructure data type.
 *
 * Created by: Fredrik Heintz 2006-07-13
 */


#ifndef ENTITY_STRUCTURE_IDL
#define ENTITY_STRUCTURE_IDL


#include "value.idl"

#pragma prefix "ida.liu.se/witas"


module Witas {
  module Idl {
    module Dyknow {

      struct EntityStructureType {
        string name;
        AttributeSeq attributes;
      };

      struct EntityStructure {
        string name;
        Time valid_time;
        Time create_time;
        AttributeValuePairSeq attributes;
      };
      typedef sequence<EntityStructure> EntityStructureSeq;

    };
  };
};


#endif
```

```
/**
 * @file entity_trajectory_structure.idl
 * Definition of the EntityStructureTrajectory interface.
 *
 * Created by: Fredrik Heintz 2006-07-13
 */


#ifndef ENTITY_TRAJECTORY_STRUCTURE_IDL
#define ENTITY_TRAJECTORY_STRUCTURE_IDL


#include "policy.idl"
#include "logable.idl"
#include "resetable.idl"
#include "entity_structure.idl"
#include "subscription_handle.idl"
#include "feature_constraints.idl"
#include "representation_exceptions.idl"

#pragma prefix "ida.liu.se/witas"


module Witas {
  module Idl {
    module Dyknow {

      /**
       * The operations common to all fstream interfaces.
       */
      interface EntityTrajectoryStructure
        : Resetable, Logable
      {
        readonly attribute Policy pol;
        readonly attribute string name;
        readonly attribute EntityStructureType type;
        readonly attribute Time start_time;
        readonly attribute Time end_time;


        EntityStructure get_latest()
          raises(ValueNotAvailable);

        EntityStructureSeq get_between(in Time from, in Time to);

        EntityStructure get_closest_at_or_before(in Time t)
          raises(ValueNotAvailable);

        EntityStructure get_at(in Time t)
          raises(ValueNotAvailable);

        EntityStructure get_at_approximating(in Time t,
                                             in ValueApproximationConstraint aprx)
```

```
      raises(ValueNotAvailable);


    SubscriptionHandle subscribe();
    void unsubscribe(in SubscriptionHandle handle);


    //void register_policy_violation_callback(in PolicyViolationCallback cb);
    //void unregister_policy_violation_callback(in PolicyViolationCallback cb);


    void add(in EntityStructure es)
      raises(OperationNotSupported);

    void add_blocking(in EntityStructure es)
      raises(PolicyViolation, OperationNotSupported);


    void create_at(in string name, in Time valid_time,
                   in AttributeValuePairSeq attributes)
      raises(OperationNotSupported);

    void create_at_blocking(in string name, in Time valid_time,
                            in AttributeValuePairSeq attributes)
      raises(PolicyViolation, OperationNotSupported);


    void create_at_now(in string name, in AttributeValuePairSeq attributes)
      raises(OperationNotSupported);

    void create_at_now_blocking(in string name,
                                in AttributeValuePairSeq attributes)
      raises(PolicyViolation, OperationNotSupported);


    /** Reset and block until finished. */
      void reset_blocking();
    };

  };
 };
};


#endif
```

```
/**
 * @file esgm.idl
 * Definition of the ESGM interface.
 */


#ifndef ESGM_IDL
#define ESGM_IDL


#include "logable.idl"
#include "pingable.idl"
#include "statable.idl"
#include "resetable.idl"
#include "memory_statable.idl"
#include "comp_unit.idl"
#include "sensor_interface.idl"
#include "esgm_exceptions.idl"
#include "policy_exceptions.idl"
#include "entity_trajectory_structure.idl"


#pragma prefix "ida.liu.se/witas"


module Witas {
  module Idl {
    module Dyknow {

      interface Esgm
        : Pingable, Resetable, MemoryStatable, Statable, Logable
      {
        /** Functionality to manage entity structure types. */
        void create_entity_structure_type(in string type_name,
                                          in EntityStructureType entity_type)
          raises(EntityStructureTypeAlreadyExists);

        void destroy_entity_structure_type(in string type_name)
          raises(NoSuchEntityStructureType);

        EntityStructureType
        get_entity_structure_type(in string type_name)
          raises(NoSuchEntityStructureType);


        /** Functionality to manage entity trajectories. */
        EntityTrajectoryStructure
        create_primitive_entity_trajectory_structure(in string name,
                                                     in string type_name)
          raises(EntityStructureAlreadyExists,
                 NoSuchEntityStructureType);

        /**
```

```
 * If from=0 then start now.
 * If to=0 continue forever.
 * If delay=0 then any delay is acceptable.
 * If cache_size=0 then the complete history is cached.
 */
EntityTrajectoryStructure
create_entity_trajectory_structure(in string name,
                                   in string sensor,
                                   in Idl::Time from,
                                   in Idl::Time to,
                                   in Idl::Time sample_period,
                                   in Idl::Time delay,
                                   in long cache_size)
  raises(EntityStructureAlreadyExists,
         NoSuchSensor, IllegalPolicy);

EntityTrajectoryStructure
create_computed_entity_trajectory_structure(in string name,
                                            in string cu,
                                            in StringSeq inputs)
  raises(EntityStructureAlreadyExists,
         NoSuchCompUnit);

EntityTrajectoryStructure
get_entity_trajectory_structure(in string name)
  raises(NoSuchEntityStructure);

EntityStructureType
get_entity_trajectory_structure_type(in string name)
  raises(NoSuchEntityStructure);

void
destroy_entity_trajectory_structure(in string name)
  raises(NoSuchEntityStructure);


/** Functionality to manage sensor interfaces. */
void add_sensor_interface(in string sensor,
                          in string type_name,
                          in SensorInterface iface)
  raises(SensorAlreadyExists);

void remove_sensor_interface(in string sensor)
  raises(NoSuchSensor);


/** Functionality to manage computational units. */
void add_comp_unit(in string name,
                   in string type_name,
                   in ValueSortSeq input_types,
                   in CompUnit cu)
  raises(CompUnitAlreadyExists);
```

```
        void remove_comp_unit(in string name)
          raises(NoSuchCompUnit);
      };

    };
  };
};


#endif
```

```
/* -*- Mode: C++ -*- */

/**
 * @file ImageWrapper.h
 * Include file for the class ImageWrapper.
 *
 * Created by: Fredrik Heintz 2006-08-09
 */


#ifndef IMAGE_WRAPPER_H
#define IMAGE_WRAPPER_H


#include "dyknow2/entity_structureC.h"
#include "dyknowutil/EntityStructureWrapper.h"


namespace Witas {
  namespace Dyknow {


    class ImageWrapper
      : public EntityStructureWrapper
    {
    public:
      ImageWrapper(const std::string& name);
      ImageWrapper(const std::string& name, long width, long height,
                   const Idl::OctetSeq& img_data, long format);
      ImageWrapper(const Idl::Dyknow::EntityFrame& ef);

      ~ImageWrapper();

      void check_attributes();


      void width(CORBA::Long val);
      CORBA::Long width() const;

      void height(CORBA::Long val);
      CORBA::Long height() const;

      void img_data(const Idl::OctetSeq& val);
      Idl::OctetSeq img_data() const;

      void format(CORBA::Long val);
      CORBA::Long format() const;

      static Idl::Dyknow::EntityStructureType type();
    };


  }
```

```
}

#endif
```

```
/* -*- Mode: C++ -*- */

/**
 * @file ImageWrapper.cc
 * Implementation file for the class ImageWrapper.
 *
 * Created by: Fredrik Heintz 2006-08-09
 */


#include "ImageWrapper.h"


namespace Witas {
  namespace Dyknow {

    ImageWrapper::ImageWrapper(const std::string& name)
      : EntityStructureWrapper(name)
    {
      set_long("width"); set_long("height");
      set_OctetSeq("img_data"); set_long("format");
    }

    ImageWrapper::ImageWrapper(const std::string& name,
                               long width, long height,
                               const Idl::OctetSeq& img_data, long format)
      : EntityStructureWrapper(name)
    {
      set_long("width", width);
      set_long("height", height);
      set_OctetSeq("img_data", img_data);
      set_long("format", format);
    }

    ImageWrapper::ImageWrapper(const Idl::Dyknow::EntityFrame& ef)
      : EntityStructureWrapper(ef)
    {
      check_attributes();
    }

    ImageWrapper::~ImageWrapper() {}

    void
    ImageWrapper::check_attributes()
    {
      if ( not exists_attribute("width", Idl::Dyknow::LONG_TYPE) ) set_long("width");
      if ( not exists_attribute("height", Idl::Dyknow::LONG_TYPE) ) set_long("height");
      if ( not exists_attribute("img_data", Idl::Dyknow::OCTET_SEQ_TYPE) ) set_OctetSeq("img_data");
      if ( not exists_attribute("format", Idl::Dyknow::LONG_TYPE) ) set_long("format");
    }

    void ImageWrapper::width(CORBA::Long val) { set_long("width", val); }
    CORBA::Long ImageWrapper::width() const { return get_long("width"); }
```

```
    void ImageWrapper::height(CORBA::Long val) { set_long("height", val); }
    CORBA::Long ImageWrapper::height() const { return get_long("height"); }

    void ImageWrapper::img_data(const Idl::OctetSeq& val) { set_OctetSeq("img_data", val); }
    Idl::OctetSeq ImageWrapper::img_data() const { return get_OctetSeq("img_data"); }

    void ImageWrapper::format(CORBA::Long val) { set_long("format", val); }
    CORBA::Long ImageWrapper::format() const { return get_long("format"); }


    Idl::Dyknow::EntityStructureType
    ImageWrapper::type()
    {
      static Idl::Dyknow::EntityStructureType img_type;
      static bool first = true;
      if ( first ) {
        img_type.name = CORBA::string_dup("ImageType");
        img_type.attributes.length(4);
        img_type.attributes[0].name = CORBA::string_dup("width");
        img_type.attributes[0].type = Idl::Dyknow::LONG_TYPE;
        img_type.attributes[1].name = CORBA::string_dup("height");
        img_type.attributes[1].type = Idl::Dyknow::LONG_TYPE;
        img_type.attributes[2].name = CORBA::string_dup("img_data");
        img_type.attributes[2].type = Idl::Dyknow::OCTET_SEQ_TYPE;
        img_type.attributes[3].name = CORBA::string_dup("format");
        img_type.attributes[3].type = Idl::Dyknow::LONG_TYPE;

        first = false;
      }
      return img_type;
    }

  }
}
```

```
/* -*- Mode: C++ -*- */

/**
 * @file esgm_tutorial.cc
 *
 * The code from the tutorial on the entity structure generation
 * module.
 *
 * Created by: Fredrik Heintz 2006-03-12
 */


#include "BlobWrapper.h"
#include "ImageWrapper.h"
#include "RobotStateWrapper.h"
#include "DrivingDirectionWrapper.h"
#include "esgm_tutorial_opt.h"

#include "dyknowutil/SeqWrapper.h"
#include "dyknowutil/SubscriptionProxy.h"
#include "dyknowutil/SubscriptionCallback.h"
#include "dyknowutil/idl_dyknow_types_extensions.h"
#include "dyknowutil/dyknow_type_conversion.h"

#include "util/OrbProxy.h"
#include "util/to_string.h"
#include "util/witasgendefs.h"
#include "util/IorServerProxy.h"
#include "util/witasiornamesdefs.h"
#include "util/get_object_from_iorserver.h"
#include "util/idl_common_types_extensions.h"

#include "dyknow2/esgmC.h"
#include "wcorba/dyknow2/comp_unitS.h"
#include "wcorba/dyknow2/sensor_interfaceS.h"

#include <ace/Thread_Manager.h>
#include <iostream>


using std::cout;
using std::endl;
using std::flush;
using std::boolalpha;
using std::string;

using Witas::Idl::Time;
using Witas::Idl::OctetSeq;
using Witas::Idl::mStringSeq;
using namespace Witas::Util;
using namespace Witas::Dyknow;
using namespace Witas::Idl::Dyknow;
```

```
/** The sensor interface to the camera. */
class CameraSensorInterface
  : public virtual POA_Witas::Idl::Dyknow::SensorInterface
{
public:
  CameraSensorInterface() {}

  virtual Sample* get_current_value(Time qtime)
    ACE_THROW_SPEC ((CORBA::SystemException));
};


/** The implementation of the sensor interface to the camera. */
Sample*
CameraSensorInterface::get_current_value(Time qtime)
  ACE_THROW_SPEC ((CORBA::SystemException))
{
  OctetSeq img_data;
  img_data.length(640*480*3);
  for ( CORBA::ULong i = 0; i < 640*480; ++i ) {
    img_data[3*i]   = 0; // red
    img_data[3*i+1] = 0; // green
    img_data[3*i+2] = 0; // blue
  }
  ImageWrapper img("image", 640, 480, img_data, 0);
  Time now = qtime;
  return new mSample(mValue(img), now, qtime, now);
}


/** The implementation of the blob detection computational unit. */
class BlobDetectionCU
  : public virtual POA_Witas::Idl::Dyknow::CompUnit
{
public:
  BlobDetectionCU() {}

  virtual Sample* compute(const SampleSeq& inputs, Time qtime)
    ACE_THROW_SPEC ((CORBA::SystemException,
                     ValueNotAvailable));
};


Sample*
BlobDetectionCU::compute(const SampleSeq& inputs, Time qtime)
  ACE_THROW_SPEC ((CORBA::SystemException,
                   ValueNotAvailable))
{
  static int counter = 0;
  //cout << "BlobDetectionCU::compute(" << inputs << ", " << qtime << ")" << endl;
  try {
    if ( inputs.length() == 1 ) {
```

```
      ++counter;
      Time now = qtime;
      ImageWrapper image(get_EntityFrame(inputs[0].val));
      // Do stuff with image
      SeqWrapper blobs("blobs");
      BlobWrapper blob("blob"+ to_string(counter), 1, 1, 10);
      blobs.push_back(blob);
      return new mSample(mValue(blobs), now, qtime, now);
    }
  } catch ( const CORBA::Exception& e ) {
    cout << "BlobDetectionCU::compute threw " << e << endl;
  }
  throw ValueNotAvailable();
}


/** The implementation of the blob selection computational unit. */
class BlobSelectionCU
  : public virtual POA_Witas::Idl::Dyknow::CompUnit
{
public:
  BlobSelectionCU() {}

  virtual Sample* compute(const SampleSeq& inputs, Time qtime)
    ACE_THROW_SPEC ((CORBA::SystemException,
                     ValueNotAvailable));
};


Sample*
BlobSelectionCU::compute(const SampleSeq& inputs, Time qtime)
  ACE_THROW_SPEC ((CORBA::SystemException,
                   ValueNotAvailable))
{
  //cout << "BlobSelectionCU::compute(" << inputs << ", " << qtime << ")" << endl;
  try {
    if ( inputs.length() == 1 ) {
      Time now = qtime;
      SeqWrapper blobs(get_EntityFrame(inputs[0].val));
      if ( blobs.size() > 0 ) {
        // Select the first blob
        BlobWrapper blob(blobs[0]);
        return new mSample(mValue(blob), now, qtime, now);
      } else {
        throw ValueNotAvailable();
      }
    }
  } catch ( const CORBA::Exception& e ) {
    cout << "BlobSelectionCU::compute threw " << e << endl;
  }
  throw ValueNotAvailable();
}
```

```
/** The implementation of the driving direction computational unit. */
class DrivingDirectionCU
  : public virtual POA_Witas::Idl::Dyknow::CompUnit
{
public:
  DrivingDirectionCU() {}

  virtual Sample* compute(const SampleSeq& inputs, Time qtime)
    ACE_THROW_SPEC ((CORBA::SystemException,
                     ValueNotAvailable));
};


Sample*
DrivingDirectionCU::compute(const SampleSeq& inputs, Time qtime)
  ACE_THROW_SPEC ((CORBA::SystemException,
                   ValueNotAvailable))
{
  //cout << "DrivingDirectionCU::compute(" << inputs << ", " << qtime << ")" << endl;
  try {
    // input[0] is the robot state and input[1] the current blob
    if ( inputs.length() == 2
         and inputs[0].vtime >= inputs[1].vtime ) {
      BlobWrapper robot_state(get_EntityFrame(inputs[0].val));
      BlobWrapper blob(get_EntityFrame(inputs[1].val));
      Time now = qtime;
      // Compute driving direction
      DrivingDirectionWrapper driving_dir("driving_dir", 10, 10);
      return new mSample(mValue(driving_dir), now, qtime, now);
    }
  } catch ( const CORBA::Exception& e ) {
    cout << "DrivingDirectionCU::compute threw " << e << endl;
  }
  throw ValueNotAvailable();
}


class SubCallback
  : public SubscriptionCallback {
public:
  SubCallback() {}
  virtual ~SubCallback() {}

  virtual void got_sample(const Symbol& sym, const Sample& val)
  {
    cout << "got sample " << sym << " = " << val << endl;
  }
};


struct TutorialData {
  int grp_id;
```

```
  bool finished;
  EntityTrajectoryStructure_var robot_state_ets;
};


void*
thread_main(void* input)
{
  TutorialData* data = static_cast<TutorialData*>(input);
  assert ( data != 0 );

  cout << "started robot sensor thread" << endl;

  while ( not data->finished ) {
    try {
      RobotStateWrapper rs("robot1", 1, 2, 3);
      data->robot_state_ets->create_at_now(rs.name, rs.attributes);
    } catch (const CORBA::Exception& e) {
      cout << "caught CORBA::Exception " << e << endl;
    } catch (const std::exception& e) {
      cout << "caught std::exception " << e.what() << endl;
    } catch (...) {
      cout << "caught unknown exception" << endl;
    }

    usleep(100*1000);
  }

  cout << "killed robot sensor thread" << endl;
  data->grp_id = -1;
  return 0;
}


int main(int argc, char* argv[])
{
  // Initialize the ORB
  OrbProxy::init(argc, argv);
  OrbProxy* orb = OrbProxy::instance();


  // Parse the rest of the arguments
  optionProcess(&esgm_tutorialOptions, argc, argv);


  try {
    // Get parameters from the command line
    int verbose = OPT_VALUE_VERBOSE;
    string ior_server_address(OPT_ARG(IORSERVER));
    Time sample_period = OPT_VALUE_SAMPLE_PERIOD*TIMEUNITS_PER_MILLI_SECOND;
    string esgm_name(OPT_ARG(ESGM_NAME));
    if ( esgm_name == "" ) {
      esgm_name = ESGM_NAME;
```

```
}

if ( verbose > 0 ) {
  cout << " verbose = " << verbose
       << "\n ior_server_address = " << ior_server_address
       << "\n esgm_name = " << esgm_name
       << "\n sample_period = " << sample_period
       << endl;
}


// Retrieve the iorserver
cout << "retrieving the ior server " << ior_server_address << "... " << flush;
IorServerProxy ior_server(ior_server_address);
cout << "done" << endl;

// Retrieve the ESGM from the iorserver
cout << "retrieving the ESGM " << esgm_name << "... " << flush;
Esgm_var esgm;
get_object_from_iorserver(&ior_server, esgm_name, esgm);
cout << "done" << endl;


// Create the entity structure types
string blob_type_name("Blob");
string image_type_name("Image");
string blob_seq_type_name("BlobSeq");
string robot_state_type_name("RobotState");
string driving_direction_type_name("DrivingDirection");

EntityStructureType blob_type;
blob_type.name = CORBA::string_dup("BlobType");
blob_type.attributes.length(3);
blob_type.attributes[0].name = CORBA::string_dup("x");
blob_type.attributes[0].type = LONG_TYPE;
blob_type.attributes[1].name = CORBA::string_dup("y");
blob_type.attributes[1].type = LONG_TYPE;
blob_type.attributes[2].name = CORBA::string_dup("size");
blob_type.attributes[2].type = LONG_TYPE;
esgm->create_entity_structure_type("Blob", blob_type);

esgm->create_entity_structure_type("Image", ImageWrapper::type());
esgm->create_entity_structure_type(blob_seq_type_name.c_str(),
                                   SeqWrapper::type(blob_type_name));
esgm->create_entity_structure_type(robot_state_type_name.c_str(),
                                   RobotStateWrapper::type());
esgm->create_entity_structure_type(driving_direction_type_name.c_str(),
                                   DrivingDirectionWrapper::type());


// Create a primitive entity trajectory structure for the robot
// sensor
string robot_state_name("robot_state");
```

```
EntityTrajectoryStructure_var robot_state
  = esgm->create_primitive_entity_trajectory_structure("robot_state",
                                                       "RobotState");


// Create an instance of the camera sensor interface
cout << "create camera sensor servant... " << flush;
CameraSensorInterface camera_sensor_iface_servant;
cout << "done" << endl;

cout << "create camera sensor corba object... " << flush;
SensorInterface_var
  camera_sensor_iface = camera_sensor_iface_servant._this();
cout << "done" << endl;

// Register the camera sensor
cout << "add the camera sensor interface... " << flush;
esgm->add_sensor_interface("camera", image_type_name.c_str(),
                           camera_sensor_iface.in());
cout << "done" << endl;

// Create an entity trajectory structure called images which
// samples the camera sensor every second forever with any delay,
// and stores the last 5 images in a cache
cout << "create images entity trajectory structure... " << flush;
EntityTrajectoryStructure_var images
  = esgm->create_entity_trajectory_structure("images", "camera",
                                             0, 0,
                                             1*TIMEUNITS_PER_SECOND,
                                             0, 5);
cout << "done" << endl;


// Create an instance of the blob detection comp unit
cout << "create blob detection comp unit servant... " << flush;
BlobDetectionCU blob_detection_cu_servant;
cout << "done" << endl;

cout << "create blob detection comp unit corba object... " << flush;
CompUnit_var blob_detection_cu = blob_detection_cu_servant._this();
cout << "done" << endl;

// Register the blob detection comp unit
cout << "add the blob detection comp unit... " << flush;
esgm->add_comp_unit("BlobDetection", blob_type_name.c_str(),
                    mValueSortSeq(ENTITY_FRAME_TYPE),
                    blob_detection_cu.in());
cout << "done" << endl;

// Create a computed entity trajectory structure called blobs which
// extract blobs from each image from the camera sensor
cout << "create blobs entity trajectory structure... " << flush;
EntityTrajectoryStructure_var blobs
```

```
  = esgm->create_computed_entity_trajectory_structure("blobs",
                                                      "BlobDetection",
                                                      mStringSeq("images"));
cout << "done" << endl;


// Create an instance of the blob selection comp unit
cout << "create blob selection comp unit servant... " << flush;
BlobSelectionCU blob_selection_cu_servant;
cout << "done" << endl;

cout << "create blob selection comp unit corba object... " << flush;
CompUnit_var blob_selection_cu = blob_selection_cu_servant._this();
cout << "done" << endl;

// Register the blob selection comp unit
cout << "add the blob selection comp unit... " << flush;
esgm->add_comp_unit("BlobSelection", blob_type_name.c_str(),
                    mValueSortSeq(ENTITY_FRAME_TYPE),
                    blob_selection_cu.in());
cout << "done" << endl;

// Create a computed entity trajectory structure called
// current_blob which selects one of the input blobs
cout << "create current blob entity trajectory structure... " << flush;
EntityTrajectoryStructure_var current_blob
  = esgm->create_computed_entity_trajectory_structure("current_blob",
                                                      "BlobSelection",
                                                      mStringSeq("blobs"));
cout << "done" << endl;


// Create an instance of the driving direction comp unit
cout << "create driving direction comp unit servant... " << flush;
DrivingDirectionCU driving_direction_cu_servant;
cout << "done" << endl;

cout << "create driving direction comp unit corba object... " << flush;
CompUnit_var driving_direction_cu = driving_direction_cu_servant._this();
cout << "done" << endl;

// Register the driving direction comp unit
cout << "add the driving direction comp unit... " << flush;
esgm->add_comp_unit("DrivingDirection",
                    driving_direction_type_name.c_str(),
                    mValueSortSeq(ENTITY_FRAME_TYPE),
                    driving_direction_cu.in());
cout << "done" << endl;

// Create a computed entity trajectory structure called
// driving_direction which computes the driving direction to the blob
cout << "create driving direction entity trajectory structure... " << flush;
EntityTrajectoryStructure_var driving_direction
```

```
  = esgm->create_computed_entity_trajectory_structure("driving_direction",
                                                      "DrivingDirection",
                                                      mStringSeq("robot_state",
                                                                 "current_blob"));
cout << "done" << endl;


// Create a data structure to be used for synchronization
TutorialData* data = new TutorialData;
data->grp_id = -1;
data->finished = false;
data->robot_state_ets
  = EntityTrajectoryStructure::_duplicate(robot_state.in());

// Create a thread which reads the robot sensor, stores the
// current values, and then sleeps for 1 second.
cout << "create robot sensor thread... " << flush;
ACE_Thread_Manager* thr_mgr = ACE_Thread_Manager::instance();
data->grp_id = thr_mgr->spawn(&thread_main, data);
cout << "done" << endl;


// Run the ORB for a while to get things started
cout << "run the orb for a while... " << endl;
ACE_Time_Value tv(3, 0);
orb->run(tv);
cout << "...finished running the orb" << endl;


// Retrieve the driving direction entity trajectory structure
cout << "get driving direction entity trajectory structure... " << flush;
EntityTrajectoryStructure_var driving_dir
  = esgm->get_entity_trajectory_structure("driving_direction");
cout << "done" << endl;

// Get the latest value
cout << "get current driving direction... " << flush;
EntityStructure_var latest = driving_dir->get_latest();
cout << *latest << endl;

/*
// Get the second latest value
EntityStructure_var second_latest = driving_dir->get_nth_latest(1);

// Get the most current value
EntityStructure_var current = driving_dir->get_latest_vtime();
*/

// Get all entity structures between t1 and t2
Time t1 = latest->valid_time - 5*TIMEUNITS_PER_SECOND;
Time t2 = latest->valid_time - 1*TIMEUNITS_PER_SECOND;
cout << "get the driving direction between " << t1 << " and " << t2 << "... " << flush;
EntityStructureSeq_var interval = driving_dir->get_between(t1, t2);
```

```
    cout << *interval << endl;

    // Approximate the value at time-point t;
    Time t = latest->valid_time - 100*TIMEUNITS_PER_MILLI_SECOND;
    ValueApproximationConstraint aprx;
    aprx._d(MOST_RECENT_VALUE_APPROXIMATION);
    cout << "get the driving direction at " << t << "... " << flush;
    EntityStructure_var at_t
      = driving_dir->get_at_approximating(t, aprx);
    cout << *at_t << endl;

    // Subscribe to the current driving direction
    cout << "subscribe to the driving direction... " << flush;
    SubCallback callback;
    SubscriptionProxy sub_proxy(driving_dir->subscribe(),
                                mSymbol("driving_dir"), &callback);
    cout << "done" << endl;


    // Run the ORB for a while, this should generate some callbacks
    cout << "run the orb for a while... " << endl;
    tv = ACE_Time_Value(2, 0);
    orb->run(tv);
    cout << "...finished running the orb" << endl;


    // Notify eventual threads that the application is finished
    data->finished = true;
    // and wait for them to finish
    cout << "wait for robot sensor thread to finish... " << flush;
    while ( data->grp_id != -1 ) {
      usleep(200*1000);
    }
    cout << "done";

    // Clean up
    delete data;
  } catch (const CORBA::Exception& e) {
    cout << "caught CORBA::Exception " << e << endl;
  } catch (const std::exception& e) {
    cout << "caught std::exception " << e.what() << endl;
  } catch (...) {
    cout << "caught unknown exception" << endl;
  }

  return 0;
}
```

Bibliography

[1] Christian S. Jensen et al. *The Consensus Glossary of Temporal Database Concepts - February 1998 Version*, 1998.

[2] Erich Rome et. al. Development of an affordance-based control architecture. Technical report, MACS/2/2.2, jun 2006.

[3] Gultekin Özsoyoglu and Richard T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Trans. Knowl. Data Eng.*, 7(4):513–532, 1995.

[4] Barry Smith. Ontology. In L. Floridi, editor, *Blackwell Guide to the Philosophy of Computing and Information*, pages 155–166. Blackwell, Oxford, 2003.